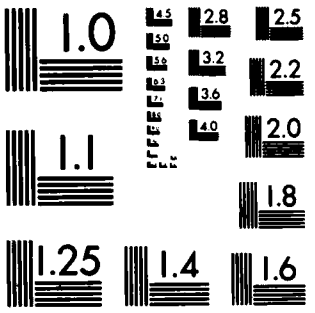
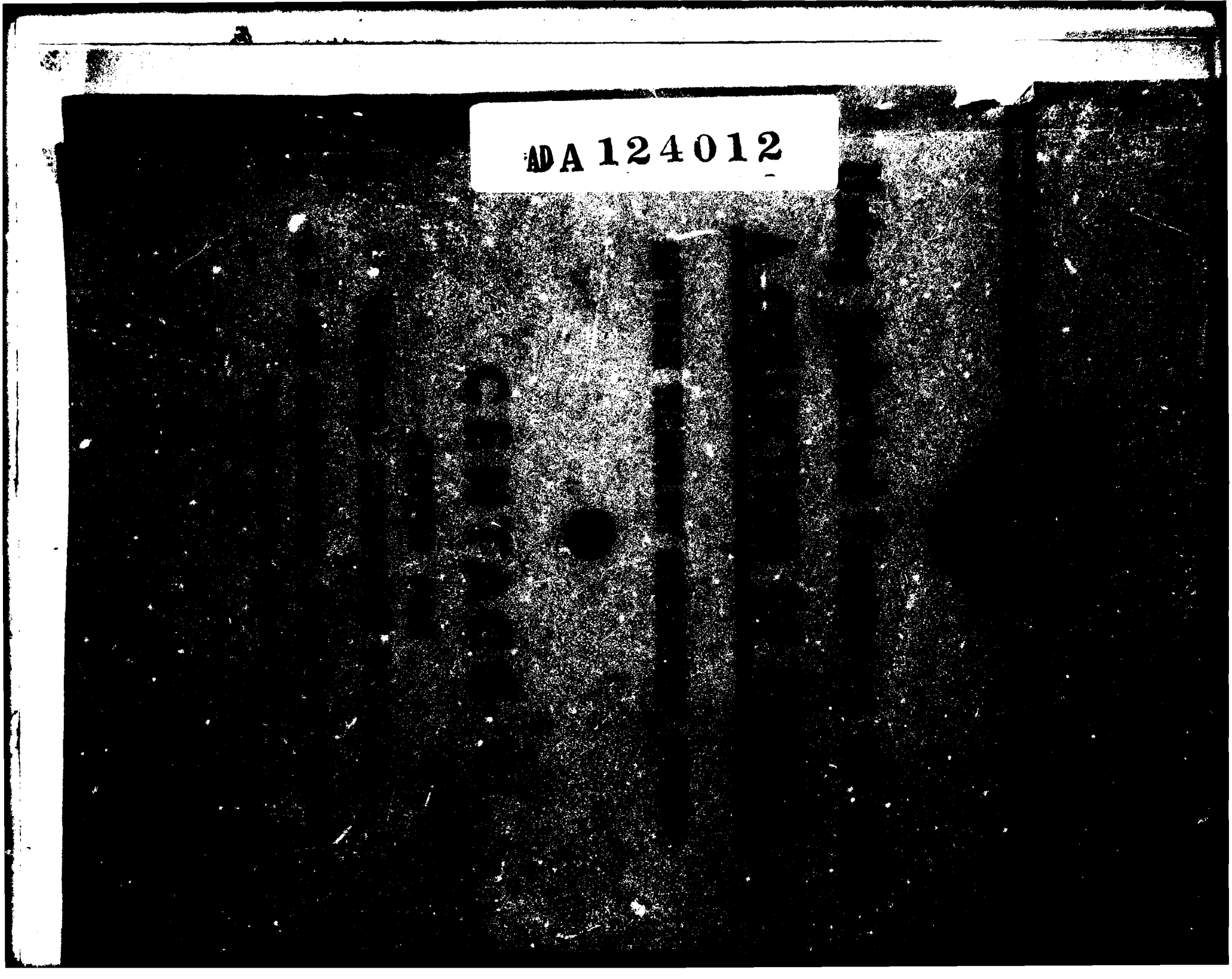
AD-A124 012 USING SELECTED FEATURES OF ADA: A COLLECTION OF PAPERS (U) BATTELLE COLUMBUS LABS OH N HABERMAN ET AL. 09 NOV 82 DAAG29-76-D-0100 1/3

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A


AD A 124012

50272-101

REPORT DOCUMENTATION PAGE	1. REPORT NO.	2. AD-A124 077	3. Recipient's Accession No.

4. Title and Subtitle
Using Selected Features of Ada: A Collection of Papers

5. Report Date
9 November 1982

6.

7. Author(s)
Nico Haberman, et al

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
USA CECOM
Center for Tactical Computer Systems (CENTACS)
ATTN: DRSEL-TCS-ADA-1
Fort Monmouth, NJ 07703

10. Project/Task/Work Unit No.

11. Contract(C) or Grant(G) No.
(C) DAAG29-76-D-0100
(G)

12. Sponsoring Organization Name and Address
USA CECOM
Center for Tactical Computer Systems (CENTACS)
ATTN: DRSEL-TCS-ADA-1
Fort Monmouth, NJ 07703

13. Type of Report & Period Covered
Final

14.

15. Supplementary Notes

16. Abstract (Limit: 200 words)

The purpose of these papers is to further the understanding of how to use selected features of the Ada Programming Language in a proper manner as viewed by the authors. Six papers are presented in this document describing the use of packages, types, tasking, exceptions, low level language features, and real data types in the Ada language.

17. Document Analysis a. Descriptors
Ada Programming Language
Ada Packages
Ada Types
Ada Exceptions

b. Identifiers/Open-Ended Terms
High Level Language
Program Design Language

c. COSATI Field/Group

18. Availability Statement:
Distribution limited to the United States. Available from National Technical Information Service, Springfield, VA 22161.

19. Security Class (This Report)
UNCLASSIFIED

21. No. of Pages
280

20. Security Class (This Page)
UNCLASSIFIED

22. Price

(See ANSI-Z39.18) See Instructions on Reverse

OPTIONAL FORM 272 (4-77)
(Formerly NTIS-35)
Department of Commerce

Using Selected Features of ADA:

A COLLECTION OF PAPERS

SOFTWARE TECHNOLOGY DEVELOPMENT DIVISION

CENTACS

CENTER FOR

TACTICAL COMPUTER SYSTEMS

U. S. ARMY COMMUNICATION - ELECTRONICS COMMAND

Fort Monmouth, New Jersey

TABLE OF CONTENTS

FOREWORD

This report is a collection of papers written for the Center for Tactical Computer Systems (CENTACS) at Fort Monmouth, NJ, under Contract No. DAAG 29-76-D-0100 (Delivery Order 1534). The authors of these papers are, respectively:

Nico Haberman - Carnegie Mellon University

John Nestor - Carnegie Mellon University

Steven Shuman - Massachusetts Computer Associates, Inc.

David Loveman - Massachusetts Computer Associates, Inc.

The final two papers were provided independently by:

Dewayne Perry - Pegasus Systems

Brian Wichmann - National Physics Laboratory, UK

The purpose of these papers is to further the understanding of how to use selected features of the Ada Programming Language in a proper manner. The viewpoints expressed in these papers are those of the authors and do not necessarily represent the viewpoint of the Army or Department of Defense.

Copies of this report may be obtained by writing to:

HQS, CECOM
Center for Tactical Computer Systems (CENTACS)
ATTN: DRSEL-TCS-ADA-1
Fort Monmouth, NJ 07703

The Use of Ada Packages[1]

A. N. Habermann
CMU, 14 November 1980

Abstract

The Ada language provides a facility for separating the specification of a program module from its implementation. The purpose of this section is to show by examples that this device greatly enhances system design.

Table of Contents

Introduction.

The Ada language provides three constructs for partitioning large programs into modules: *packages*, *tasks* and *subprograms*. In this essay we will not treat subprograms as a separate construct, because the subprogram construct is in fact nothing else than a special case of the package construct. Normally, a package exports several functions and procedures. A subprogram is a package that exports a single function or procedure. A package is typically used for grouping data types and operations on objects of those types into a single unit. Tasks serve a similar purpose, but also define the execution order of operations. In this first essay we focus our attention on the organization of sequential programs. Tasks are discussed in a separate section in this document.

1 Package Specifications.

An Ada package consists of a *visible part* and, if needed, a *package body*. The visible part specifies program objects such as constants, types and subprograms, while the package body describes their implementation (see Ref.Man. Chapter 7: "Modules"). An example of a Package Specification is:

```
package IntegerPairs is

    type pair is record xcomp , ycomp : integer; end record;

    function assemble       (p, q : in integer)          return pair;
    function  " + "         (p, q : in pair)             return pair;
    function  "-"           (p, q : in pair)             return pair;
    function  "*"           (c : in integer, q : in pair) return pair;
     -- this function defines scalar multiplication of the form "scalar * pair"

end IntegerPairs;
```

This visible part specifies the program objects that this package provides to its surrounding scope (see Ref.Man. Section 8.1: "scope of Declarations"). It defines the interface of this module with other program modules and lists the program objects that it creates and that can be used by other modules. Type "pair" is an example of an *open type*, which makes the structure of objects of this type visible to its users. The functions are specified in sufficient detail for users to write operations on pairs and integers, but the implementation remains hidden from the users. The implementation is described in a separate piece of program, the body of **package** IntegerPairs.

```
package body IntegerPairs is
    <<< programs for functions assemble, " + ", "-" and "*" >>>
end IntegerPairs;
```

It is not necessary that the implemetation body immediately follows the visible part of a program module. Placing several visible parts together has the advantage that one can oversee the structure of an entire subsystem and the facilities provided by its various modules. Having to look at implementations is often confusing when the overall structure of a system is the issue.

The example shows how in Ada programs the *specification* of a program module is separated from its implementation. In most other languages, a programmer is forced to interleave specifications and implementations, and put them into a single module. Not separating the two causes problems for both users and system designers. Without the separation, a user must search through the code of a program module and find the parts relevant to his application. In this situation it is essentially up to the user to decide which pieces of code he should consider as specification and which pieces he should ignore as implementation detail. Such decisions are not only error prone, but often lead to undesirable implementation dependencies between program modules. Such dependencies are not explicitly visible in the program text and make it therefore extremely difficult to modify programs without introducing new errors that are very hard to discover. The strict separation of specification and implementation in Ada does not leave it up to the user to decide which piece of code is relevant. The distinction is explicit in programs so that implementation details can be hidden from the users of a program module.

For system designers the drawback of mixing specifications and implementations is that one is inclined to go forwards and backwards between design and writing code, instead of concentrating on one or the other. We will show in this section that in Ada system design and writing code are very distinct activities. The natural thing to do in Ada is first to write specifications for a collection of interrelated program modules and then to write programs that implement these separate modules at some later time. This simple device of partitioning a module in a specification part and an implementation body is a powerful support tool for system design.

2 Design by Similarity.

A common case of using package specifications is that of designing new packages by similarity. We discuss two different ways of exploiting similarity:

1. design packages similar to an existing package;

2. design a template and introduce new packages as instantiations of that template.

We illustrate the ideas by introducing fractions, integer vectors and complex integers. The obvious thing to do is to write three packages similar to the one for IntegerPairs.

```
package Fractions is

    type fraction is record numer, denom : integer; end record;

    function assemble       (p, q : in integer)              return fraction;
    function "+"            (p, q : in fraction)             return fraction;
    function "-"            (p, q : in fraction)             return fraction;
    function "*"            (c : in integer, q : in fraction) return fraction;
      -- this function defines scalar multiplication of the form "scalar * fraction"
end Fractions;


package IntegerVectors is

    type vector is record xcomp , ycomp : integer; end record;

    function assemble       (p, q : in integer)              return vector;
    function "+"            (p, q : in vector)               return vector;
    function "-"            (p, q : in vector)               return vector;
    function "*"            (c : in integer, q : in vector)  return vector;
      -- this function defines scalar multiplication of the form "scalar * vector"
end IntegerVectors;


package ComplexIntegers is

    type plexint is record re , im : integer; end record;

    function assemble       (p, q : in integer)              return plexint;
    function "+"            (p, q : in plexint)              return plexint;
    function "-"            (p, q : in plexint)              return plexint;
    function "*"            (c : in integer, q : in plexint) return plexint;
      -- this function defines scalar multiplication of the form "scalar * plexint"
end ComplexIntegers;
```

The specification of all three packages is exactly the same as that of IntegerPairs. It is obvious that the implementation of the three functions will be the same as that of IntegerPairs for packages IntegerVector and ComplexIntegers, but not for Fractions. In the latter case addition and subtraction should not be applied component-wise, while scalar multiplication should have an effect on the numerator, but not on the denominator. IntegerVectors and ComplexIntegers can both use the scalar multiplication of IntegerPairs which multiplies both components of a pair or vector or complex number. In case the specification and implementation are the same (as is the case for IntegerPairs, IntegerVectors and ComplexIntegers) there is no need to write three separate packages. If package IntegerPairs has been defined, all that is needed instead of two new packages is the pair of declarations:

```
type IntegerVector is new pair;
type ComplexInteger is new pair;
```

This declaration validates the operations defined in package IntegerPairs also for objects of type IntegerVector and of type ComplexInteger (see Ref.Man. 3.4 "Derived Type Definitions"). For example,

```
declare   x, y : ComplexInteger;
          p, q : pair;
begin
          x := assemble(3, 4);
          y := 2 * x + 3 * assemble(5, 12);
          p := assemble(3, 4);
           -- q := p + x;  is incorrect => type violation.
----------
end;
```

Note that one can derive one type from the other if and only if the operations defined for those types are similar in the parameters they use and also in the way they work. Derived types not only use the same *specifications*, but also the same *implementations*. The example of Fractions serves the purpose of showing that packages may have similar specifications (have similar visible parts), but cannot be derived from one another because of the different implementations (that must be written as different package bodies). Packages IntegerPairs, IntegerVectors and ComplexIntegers can share a common implementation body, but that of Fractions is not the same.

Instead of deriving IntegerVectors and ComIplexIntegers from package IntegerPairs, one can first define a package *template* that contains all the common elements of IntegerPairs, ComplexIntegers and IntegerVectors. The three packages can then be defined as instantiations of this template.

A package template is defined by a *generic package* which has the same format as an ordinary package except for a prefixed *generic clause* (see Ref.Man. 12.1, "Example of a generic package declaration"). A generic clause is frequently used to introduce a formal type parameter. This is particularly useful if one wants to write a piece of program that should work for a variety of types. A well-known example is that of a "stack" for which one defines the operations "push" and "pop". One would like to define a stack package independent of the stack element type so that one can introduce stacks of reals, stacks of records or stacks of access variables without having to rewrite the code for "push" and "pop" (see Ref.Man. 12.4).

We try to capture the commonality of IntegerPairs, IntegerVectors and ComplexIntegers by defining a generic package "Pairs". The following definition is a first step in that direction, although the generic specification is not complete.

```
generic type comp is private;  -- this generic clause is incomplete

package Pairs is

    type pair is record xcoord, ycoord : comp; end record;

    function assemble     (p, q : in comp)              return pair;
    function "+"          (p, q : in pair)              return pair;
    function "-"          (p, q : in pair)              return pair;
    function "*"          (c : in comp, q : in pair)    return pair;
      -- this function defines scalar multiplication of the form "scalar * pair"
end Pairs;
```

The qualification *private* in the formal type definition means that the user can supply any structured type he wants and that the generic package will not assume any particular structure. The generic package will treat formal type "comp" and objects of that type as a black box. This implies a restriction for the generic package in that it cannot directly access the structure of objects of the formal type. That excludes all array or record access to those objects in the code of the generic package.

The information that is missing in the generic clause has to do with operations that can be performed on objects of the formal type "comp". When the functions " + ", "-" and "*" are implemented, their code uses the fact that elements of pairs can be added, subtracted and multiplied. It must be clear to users of the generic package that the actual type he supplies when he uses the generic package must be one for which addition, subtraction and multiplication are defined. It would for instance make no sense to use type "stack" as the actual type for "comp", because addition, subtraction and scalar multiplication do not make sense for stacks. The correct definition of generic package "Pairs" in Ada is:

```
generic type comp is private;
      with function " + " (comp, comp) return comp;
      with function "-" (comp, comp) return comp;
      with function "*" (comp, comp) return comp;

package Pairs is

      type pair is record xcoord, ycoord : comp; end record;

      function assemble      (p, q : in comp)              return pair;
      function " + "         (p, q : in pair)              return pair;
      function "-"           (p, q : in pair)              return pair;
      function "*"           (c : in comp, q : in pair)    return pair;
        -- this function defines scalar multiplication of the form "scalar * pair"
end Pairs;
```

Generic package "Pairs" can be used to define IntegerPairs, two-dimensional vectors and complex numbers. The use of a generic package has the additional advantage that we can define vectors of various component types if we want to.

```
package IntegerPairs      is new Pairs (comp is integer);

package Vectors           is new Pairs (comp is digits 7);

package Complex           is new Pairs (comp is digits 9);

package ComplexVectors    is new Pairs (comp is Complex.pair);
```

The latter defines two dimensional vectors of complex numbers. Note that "pair" as declared in **package** Pairs by itself is not a type, but a *template* for a type. When **package** Pairs is instantiated, as in the definitions above, a new type is created by making a copy of the declared template. The four definitions above introduce four distinct types: IntegerPairs.pair, Vectors.pair, Complex.pair and ComplexVectors.pair.

Other operations on these types of objects can be defined in additional packages. For example,

```
package ComplexOps is

    use Complex;

    function "*"        (p, q : in pair)    return digits 9;    -- inner product
    function modulus    (p : in pair)       return digits 9;

end ComplexOps;
```

The name *pair* in this example means Complex.pair, the type that is defined as part of **package** Complex (see Ref.Man. Section 8.4 "Use Clauses").

Note that, again, a package Fractions cannot be defined with what we now have. The same problem arises as before because of the different implementations needed for the operations on fractions. A generic package body satisfying the needs of complex numbers and vectors does not provide the correct implementation for fractions. No matter how similar the specifications, it is necessary to define a separate package for Fractions.

```
package Fractions is

    type fraction is record numer, denom : integer; end record;

    function assemble   (a,b : in integer)      return fraction;
    function " + "      (p, q : in fraction)    return fraction;
    function "-"        (p, q : in fraction)    return fraction;
    function "*"        (p, q : in fraction)    return fraction;
    function recip      (p : in fraction)       return fraction;

end Fractions;

package body Fractions is

    <<<implementation of " + ", "-", "*", assemble and recip for fractions>>>

end Fractions;
```

The examples of this section show that one can go through a fair amount of design without considering implementation details. It is typical for a designer to come up with an initial design and then consider revisions according to the insight gained by doing the initial work. This section demonstrated the use of package specifications for expressing such design decisions.

3 Specialization of General Packages.

Another common case is that of designing special versions of a given package. If the latter provides a collection of primitive types and basic operations, its application may be inconvenient and cumbersome in specific cases. It is for instance undesirable to force users to make use of general file facilities for terminal I/O. It is customary that the programming environment automatically creates access to screen and keyboard without requiring the user to open files for these devices. The programming environment also makes it unnecessary for users to indicate in every read or write operation that screen or keyboard is used as I/O device. One expects to find special versions for reading the keyboard and writing the screen. The purpose of this section is to show the use of Ada packages for introducing specialized versions of a given set of primitive facilities.

The goal of the following exercise is to design special versions of a general mailing system. Let us assume that *messages* are defined by a **package** MSG:

```
package MSG is

    subtype amount is integer range 0 . . integer'last;
    type content(size : amount);                    -- incomplete type declaration
    type message is access content;
    type content(size : amount) is
        record
                sender : string(1 . . 12);
                  date : string(1 . . 8);
                  text : string(1 . . size);
                  next : message;
        end record;

end MSG;
```

The need for an incomplete type declaration is explained in Ref. Man. 3.8, "Access Types". Messages can be created dynamically by assigning the designator returned by allocator "**new** content(something)" to a variable of type message. For example, a collection of five messages of equal size is generated by the subprogram:

```
declare x : message := new content(120);
begin
    for q in 1 . . 4 loop
        x.next := new content(120); x := x.next;
    end loop;
        ----------
end;
```

A general mail facility is described by the visible part of package MailSystem:

```
package MailSystem is
    use MSG;

    type mailbox is private;

    procedure   deposit   (m : in message; box : inout mailbox);
                                -- add new last message to box
    procedure   receive   (m : out message; box : inout mailbox);
                                -- take first message out of box
    procedure   clear     (date : in string; box : inout mailbox);
                                -- remove all messages up to "date"
    procedure   remove    (sender : in string; box : inout mailbox);
                                -- remove all messages from "sender"

    function    boxsize   (box : in mailbox) return MSG.amount;
    function    lookup    (sender : in ; box : in mailbox) return mailbox;
                                -- duplicate all messages from user in new box

private
    type mailbox is record first, last : MSG.message; end record;
end MailSystem;
```

The first special function is one for a mail system defined for a static number of users. Each user has exactly one mailbox which exists as long as the user wants. Since every user has a unique mailbox, it is no longer necessary to address a particular mailbox. A *userID* can be used instead. The package body of the special version we are designing manages an array of mailboxes and selects a particular mailbox by using a userID as index. (The example is somewhat simplistic, because no protection is built in against users that make unauthorized use of somebody else's userID.)

```
package StandardMail is
    use MSG;

    maxuser : constant : = 81;
    type userID is new integer range 0 . . maxuser;

    function    getID                                          return userID;
    procedure   releaseID   (x : in userID);
    procedure   deposit     (m : in message; user : in userID);
    function    receive     (owner : in userID)                return message;
    procedure   clear       (date : in string; owner : in userID);
    function    mesnum      (user : in userID)                 return amount;
    function    lookup      (owner, user : in userID)          return message;
    function    nextmsg     (m : in message; owner : in userID) return message;

end StandardMail;
```

Another example of a special application is that of a "suggestion box" used for collecting ideas from employees for improving the working environment or general operations. In this case the package body of the special version we are designing declares only one unique mailbox which it manages internally. (The package provides a procedure for deleting messages from the suggestion box. We assume that the management of the company will take care that this procedure is not misused.)

```
package SuggestionBox is
    use MSG;

    type suggestion is new message;

    procedure   suggest        (m : in suggestion);
    procedure   delete         (date : in string);
    function    firstsuggestion                        return suggestion;
    function    nextsuggestion (m : in suggestion)     return suggestion;
end SuggestionBox;
```

The new type *suggestion* has been derived from type *MSG.message*. The mechanism of derived types is explained in Ref.Man. 3.4. It is used in this example to remind the user that he is handling an object that came from - or that will be sent to - the SuggestionBox.

One can think of other special applications such as UNIX pipes, a personal appointments calendar, etc. The point of the exercise was to demonstrate the use of packages as a tool for specifying special applications of a general facility. Using packages, one can easily create modified versions of given facilities and hide irrelevant details.

4 Packages as Data Types.

Packages are often used for introducing a data type and the set of operations that apply to objects of that type. All packages discussed so far do exactly that. This section shows that in some cases the package definition itself can serve as the definition of a data type.

Ada distinguishes three classes of types: *open types*, *private types* and *limited private types*. The latter two restrict the access to objects of such a type in certain ways. What all three have in common is that users can apply the operations that are defined in the visible part containing the type declaration. The differences are in structure access operations, such as array access and record field access, and in assignment or equality tests.

Open types are those whose structure is displayed in the visible part of a package. An example is type *message* in package *MSG* (Section 3.) A user of the type has access to the structure and can apply record or array access operations to objects of that type (whichever is appropriate). Assignment and equality tests are also allowed for open types.

Private types hide their structure from users (see Ref.Man 7.4.1). An example of a private type is **type** MailBox in **package** MailSystem (see Section 3). In this case a user cannot apply structure access operations, but assignment and equality tests are allowed.

Limited private types behave like private types in that structure access operations are not permitted to objects of such a type. In addition, assignment and equality tests are also not permitted. Examples of types for which such restrictions make sense are Stack, Queue, Buffer, etc. In each of these cases it makes little sense to overwrite one of those objects with the content of another one (of the same type). The concept of limited private type is introduced in Ref.Man. 7.4.2.

In the case of a limited private type one can often omit the type definition entirely and define a generic package instead that plays the role of a limited private type. The point is illustrated by designing a package for queues. The operations typically defined for queues are ENQ and DEQ, which make it possible to put items into a queue at one end and take items out at the other end. We first look at a **package** QUE that contains an explicit **type** declaration for queues, and then at one that contains no explicit **type** declaration for queues.

With an explicit type declaration for queues, the definition of **package** QUE looks like this:

```
generic
      qsize : integer range 1 . . 64;
      type T is private;

package QUE is

      type queue is limited private;

      procedure   ENQ     (item : in T; q : inout queue);
      procedure   DEQ     (item : out T; q : inout queue);

private
      type queue is
                  record
                        content : array ( 1 . . qsize ) of T;
                        front, size : integer range 0 . . qsize := 0;
                  end record;

end QUE;
```

It is obvious that one wants to define package QUE as a generic package so that one can declare queues of different sizes and queues containing elements of various types. Note the different meaning of the keyword *private* in three places. In the generic clause it means that **package** QUE will not make assumptions about type T and will not access the structure of objects of type T. In the declaration of type queue it means that *users* of **package** QUE cannot access the structure of queues. The **private** section at the end of the visible part displays the implementation of **type** queue to an Ada compiler (see Ref.Man.7.2 "Package Specifications and Declarations").

Instead of declaring **type** queue explicitly, we now define that type implicitly as part of the definition of **package** QUE.

```
generic
      qsize : integer range 1 . . 64;
      type T is private;

package QUE is

      procedure   ENQ     (item : in T);
      function    DEQ                    return T;

end QUE;
```

The package body of QUE contains local declarations for the queue body and for the variables that keep track of the front and the size of the queue.

```
package body QUE is
    front, size : integer range 0 . . qsize : = 0;
    qbody : array ( 1 . . qsize ) of T;

    procedure   ENQ   is          ----------          end ENQ;
    function    DEQ   is          ----------          end DEQ;

end QUE;
```

If a user wants to create a queue of a particular size for a particular type of elements (for complex numbers for instance), he writes in his program the declaration:

```
package PlexQue is new QUE(qsize = > 36, T = > Complex.pair);
```

There may be many similar declarations in a program that each introduce a new queue. Operations on the example queue are denoted by "PlexQue.ENQ(u)" and "PlexQue.DEQ", where "u" is a variable or expression of type Complex. A similar example is found in Ref.Man. 12.4.

It is a good idea to define a type implicitly through a package if one wants to generate isolated objects that are not used in conjunction with one another. One should realize that generic packages are somewhat more restricted than limited private types, because instances of packages cannot be passed as parameters to subprograms. The fact that such basic operations as assignment and equality tests are not permitted for limited private types implies that one is probably also not interested in passing objects of limited type as parameters. Stacks, queues, buffers and the like are typical examples of objects for which definition as a generic package is appropriate. Complex numbers, and in general objects that are treated as part of collections, should not be defined as instantiations of a generic package. It would not be possible to write operations on complex numbers that use parameters of type complex. Limited private types form a class of types for which passing objects of such a type as parameters is often unnecessary. The objects themselves are hardly manipulated in their entirety, which is the main reason for not permitting assignment and equality tests. In this case objects are often used in isolation from one another, so an implicit type definition through a package makes sense.

TYPES

by

John Nestor

Table of Contents

1. Types

This essay discusses a central part of the Ada language: its type system. The type system controls all data declaration and manipulation in Ada, and a reasonably complete understanding of it is necessary for all programmers who use Ada. The discussion is divided into three sections. Section 1.1 discusses the basic Ada rules for data and types. Section 1.2 considers how programs can be modularized using the abstract type concept. Section 1.3 discusses some techniques that can be used to generalize type definitions so that types can be composed to form other types. Throughout the discussion suggestions are made on how to use Ada types in an effective manner. A key part of this is the presentation of techniques for producing maintainable and machine-independent programs.

1.1 Type Structure

The major purpose of types is to structure data within a program and to enforce properties that ensure the consistency of that data.

1.1.1 Objects, Types, and Subtypes

The basic unit of data in Ada is an *object*. Variables, constants, and formal parameters (when bound to actual parameters during a call) are all objects. Each object has a set of properties that control what values the object may have and what operations can be applied to the object. A major purpose of these properties is to enforce consistent use of the object by means of appropriate checking. Some properties are determined at compile time (that is, they are completely checked during the compilation process), while other properties are determined at run time (that is, they are checked only when a section of code that depends upon them is actually executed). A run time check that succeeds during one execution of the code section may fail during some other execution in which the objects have different values.

The properties that an object has are its type, its subtype, and its value. These properties differ in the time at which they are determined. The *type* of an object is determined at compile time. It controls the general structure of the values that an object can have. For example, the type states whether an object can have boolean values, or integer values, or some specific kind of structured value. The type is used for such compile time operations as type checking and overload resolution. The *subtype* of an object is determined when the object is created at run time. The subtype of an object consists of its type together with a set of *constraints* that are specific to the type. The constraints of the subtype of an object serve to further limit the permitted values; however, in this case the limits can depend upon run time computations. For example, the range constraint of an integer

subtype limits the values of objects with that subtype to some run time determined contiguous finite range of integer values. An array subtype includes an index constraint which determines the number of elements in the array. Finally, each object has a *value* which can, in general, be changed by assignment anytime during the lifetime of the object.

Since, in general, there will be several objects that share the same type and/or subtype, the language provides declarations for both types and subtypes that can then be referenced in other declarations. For example

```
type Int is range -1024..1024;

subtype NInt is Int range 0 ..1024;

I1,I2: Int;
N: NInt;
```

In some cases, the Ada type declaration is actually used to declare not only a type, *but also a subtype*. For example

```
type Bit10 is array (Int range 1..10) of boolean;
```

is actually equivalent to

```
type Bit10T is array (Int range <>) of boolean;
subtype Bit10 is Bit10T(1..10);
```

where Bit10T is a compiler generated name distinct from all other names that appear in the program. When using the type declaration in this way the user should be careful to understand which of the constraint-like specifications are part of the type and which are part of the subtype. For example the user can write

```
type Bit_I_J is array (Int range I..J) of boolean;
```

where I and J are variables with type Int. This is permitted because here the range is part of the subtype. However the user may not write

```
type I_J is range I..J;    -- This is illegal
```

where I and J are again Int variables. This is illegal because the range here determines the *type* and must therefore be known at compile time. It would be ideal if it were always possible for the user to use the type declaration only to create types; however, this is not always possible. For example

```
type Vector3 is array (Int range 1..3) of Int range 0..100;
V3: Vector3;
```

defines both a type and a subtype, but here there is no way to rewrite this as a type declaration that declares only a type and a subtype declaration that specifies the constraints. The reason for this is that given an array type there is no way to specify further subtype constraints that apply to the array element type. When this form is used, only a single subtype of the declared type can ever be used, because there is no way to refer to other subtypes or to the type itself. Further problems can arise because the subtype of an array slice, such as V3(2..3), can not be referenced. When more that one

subtype of the type is needed in these cases, then the techniques discussed in the subsection on parameterized types can sometimes be used.

The subtype declaration can also be used to further constrain a previously declared subtype.

```
subtype T1 is Int range 0..100;
subtype T2 is T1 range 1..100;
```

This technique should only be used when T2 is logically related to T1 (via some further constraint). When this is not true, then it would be better to write

```
subtype T1 is Int range 0..100;
subtype T2 is Int range 1..100;
```

When this form is used, changes to T1 will have no effect on T2.

1.1.2 Naming

In Ada, all types are named. Most types are named by the user, but there are also *anonymous* types whose names are assigned by the compiler. Type naming is the basis of the Ada type equivalence rules. Two types are *equivalent* if and only if they have the same name. The major use of type equivalence is in type checking, which occurs during compilation. A value of an object can be assigned to some other object only if the two objects have equivalent types, or more simply if the types have the same name. Similarly an actual parameter can only be passed to a formal parameter with an equivalent type.

Type equivalence also plays a key role in the overloading of subprograms. Overloading is permitted for two procedures (or functions) with the same name and with the same number of parameters if the types of at least one pair of corresponding formal parameters (or for functions result types) are not equivalent.

The type declaration serves to introduce a new named type. In the case of composite types, each component type (including the index type for arrays) should have a separate type declaration. For example

```
type I10 is range 1..10;

subtype Index is Int I..J;

type R is record
      X,Y: I10;
   end record;

type P is access R;

type A is array (Index) of P;

VR1,VR2: A;
K: Index;
```

When components are now selected, they will each have a type whose name has been declared.

```
VR1             -- has type A

VR1(K)          -- has type P

VR1(K).all      -- has type R

VR1(K).X        -- has type I10
```

Because each component type is named, it will be possible to declare a variable which can hold any component value and it will be possible to pass any component as a parameter.

It is also possible to have anonymous types whose type name is assigned by the compiler. Each anonymous type is different from all other types, whether user named or anonymous. Anonymous types should normally be avoided, since they will severely restrict what can be done with objects or components that have that type. For example

```
Anon: array (Index) of boolean; -- Bad programming style
```

Here it will not be possible to pass Anon as a parameter since its type can not be referenced by the user. Even if the object is never passed as a parameter, anonymous types should still be avoided. This will avoid difficulties when the program is modified and the programmer discovers that an object with an anonymous type then needs to be passed as a parameter.

Although in most cases anonymous types should be avoided, there are two places where they can reasonably be used. The first place has already been discussed. Recall the example

```
type Vector3 is array (Int range 1..3) of Int range 0..100;
```

which declared both a type and its only subtype. In this case the type name is anonymous; however this causes no problems since there is a name by which the only subtype of that type that is ever used can be referenced. The other place where anonymous types are needed is discussed in the subsection on parameterized types.

1.2 Abstract Types

As was discussed in the chapter on program structure, packages are a key tool for achieving program modularity. In this section the use of packages to produce abstract types is considered. As a basis for discussion the following abstract type will be used. Various techniques used in this example will be discussed throughout this and later sections.

```
package Stacks is

   type StackSizeT is range 0..10000;
   StackMax: constant StackSizeT:= 100;

   exception Overflow,Underflow,Post_Failure;

   type ElemT is range -100..100;
   type Stack is limited private;

   procedure Init(S: out Stack);

   procedure Push(S: in out Stack;I: in ElemT);
   -- can raise Overflow

   procedure Pop(S: in out Stack);
   -- can raise Underflow

   function Top(S: in Stack) return ElemT;
   -- can raise Underflow

   procedure Final(S: in out Stack);
   -- can raise Post_Failure


private

   type ElemArray is array(StackSizeT range 1..StackMax) of ElemT;
   type Stack is record
         Top: StackSizeT range 0..StackMax;
         Elems: ElemArray;
      end record;

end Stacks;


package body Stacks is

   procedure Init(S: out Stack);
   begin
      S.Top:= 0;
   end Init;
```

```
        procedure Push(S: in out Stack;I: in ElemT);
        begin
           if S.Top = StackMax then
              raise Overflow;
           end if;
           S.Top:= S.Top + 1;
           S.Elems(S.Top):= I;
        end Push;


        procedure Pop(S: in out Stack);
        begin
           if S.Top = 0 then
              raise Underflow;
           end if;
           S.Top:= S.Top - 1;
        end Pop;


        function Top(S: in Stack) return ElemT;
        begin
           if S.Top = 0 then
              raise Underflow;
           end if;
           return S.Elems(S.Top);
        end Top;


        procedure Final(S: in out Stack);
        begin
           if S.Top /= 0 then
              raise Post_Failure;
           end if;
        end Final;

     end Stacks;
```

Note that the visible part includes, as comments, information about which exceptions can be raised by each visible subprogram. It is also usually desirable to include a comment with each visible declaration that describes its use (i.e. for a subprograms this comment would describe its effect when called).

In this example there are two special visible procedures, Init and Final, which are to be called explicitly by the user at the beginning and end of the lifetime of any Stack object. For example

```
declare
   S: Stack;
begin
   Init(S);
   ...      -- All other uses of S occur here
   Final(S)
end
```

The Init procedure serves to establish any needed preconditions for S (i.e. it makes sure that S is initially in a consistent and useful state). The Final procedure is used to check any desired postconditions (i.e it ensures that S has been left in a reasonable state) and in some cases to do various clean-up operations just before the end of the lifetime of S.

One problem with both Init and Final is that the user must always remember to call them explicitly at the beginning and end of the lifetime of any stack object. If the user forgets to make these calls, then unexpected behavior may result. In the case of Init it is possible to ensure that it is automatically called whenever a Stack object is declared (unfortunately there is no way to cause Final to be automatically called). This can be done most simply by deleting the Init routine and changing the Stack type declaration to be

```
type Stack is record
      Top: StackSizeT range 1..StackMax:=0;
      Elems: ElemArray;
   end record;
```

Although this approach works for Stack, there are abstract types that require more general initialization than can be achieved via the simple initialization of record components. In this case a more powerful (and unfortunately less efficient) approach must be used. Using the original Stack example as a starting point three changes are needed. First, the Stack type declaration is changed to be

```
type Stack is record
      Initialized: boolean:= false;
      Top: StackSizeT range 1..StackMax;
      Elems: ElemArray;
   end record;
```

Second, the specification of Init is removed from the visible part and in the body part it is changed to be

```
procedure Init(S: in out Stack);
begin
   if not S.Initialized then
      S.Initialized:= true;
      S.Top:= 0; -- Arbitrary initialization code can go here
   end if;
end Init;
```

Finally, a call to Init is placed as the first statement in the body of the Push, Pop, Top, and Final subprograms. The first of these calls to done will do the initialization, while in later calls the if test in Init will fail.

1.2.1 Representation Hiding

One of the most important characteristics of abstract types is that their representation is hidden. This means that users of the abstract type need not understand irrelevant representational details. Even more important the representation of an abstract type can be changed providing its abstract behavior remains unchanged. Since changing the representation of a data structure is one of the most frequent kinds of program changes that are made to improve program performance, representation hiding becomes crucial to program maintenance. The main tool for the hiding of representations is private types. All abstract types should be private. This hides most representational details: however, the automatically defined assignment and equality routines for a private type can allow certain aspects of the representation to be visible to users of the abstract type. This can be seen by comparing a stack with an array representation with another stack that uses a linked list representation. The automatically supplied assignment for an array will copy the array while the automatically supplied assignment for the linked list will copy only the pointer to the start of the list. The user can now detect the difference in representation as shown here:

```
S1,S2:Stack;
...
Push(S1,1);
Push(S2,2);
S1 := S2;  - here is where the problem happens!
Push(S1,3);
if Top(S2) = 3 then
   -- pointer assignment
else
   -- copy assignment
end if;
```

Similar detection of representation is also possible by using the automatically supplied equality. One way to avoid this problem is to make all abstract types be limited private. Since assignment and equality are not automatically provided in this case, all representational details are hidden. Another approach that will also work is to always use an access type as the top level of the representation of an abstract type. However, there is extra overhead associated with this use of pointers and pointers will severely limit the optimizations that most Ada compilers will be able to detect; therefore, this approach is not recommended. The example below shows how Stack can be changed to use a linked list rather than an array representation. This example is abstractly equivalent to the original Stack example.

```
with UncheckedDeallocation;
package Stacks is

   type Stack is limited private;

   type StackSizeT is range 0..10000;
   StackMax: constant StackSizeT:= 100;

   exception Overflow,Underflow,Post_Failure;

   type ElemT is range -100..100;
   type Stack is limited private;

   procedure Init(S: out Stack);

   procedure Push(S: in out Stack;I: in ElemT);
   -- can raise Overflow

   procedure Pop(S: in out Stack);
   -- can raise Underflow

   function Top(S: in Stack) return ElemT;
   -- can raise Underflow

   procedure Final(S: in out Stack);
   -- can raise Post_Failure


private
   type ItemT;
   type ItemPtr is access ItemT;
   type ItemT is record
         Val: ElemT
         Next: ItemPtr;
      end record;
   type Stack is record
         Top: StackSizeT range 0..StackMax;
         Elems: ItemPtr;
      end record;

end Stacks;
```

```
procedure Free is new UncheckedDeallocation(ItemT,ItemPtr);

package body Stacks is

   procedure Init(S: out Stack);
   begin
      S.Top:= 0;
      S.Elems:= null;
   end Init;


   procedure Push(S: in out Stack;I: in ElemT);
   begin
      if S.Top = StackMax then
         raise Overflow;
      end if;
      S.Top:= S.Top + 1;
      S.Elems:= new ElemPtr(I,null);
   end Push;


   procedure Pop(S: in out Stack);
      P: ItemPtr;
   begin
      if S.Top = 0 then
         raise Underflow;
      end if;
      P:= S.Elems;
      S.Elems:= P.Next;
      Free(P);
      S.Top:= S.Top - 1;
   end Pop;


   function Top(S: in Stack) return ElemT;
   begin
      if S.Top = 0 then
         raise Underflow;
      end if;
      return S.Elems.Val;
   end Pop;


   procedure Free_ItemPtr(P: in ItemPtr)
   begin
      if P /= null then
         Free_ItemPtr(P.Next);
         Free(P);
      end if;
   end Free_ItemPtr
```

```
    procedure Final(S: in out Stack);
    begin
       Free_ItemPtr(S.Elems);
       if S.Top /= 0 then
          raise Post_Failure;
       end if;
    end Final;

end Stacks;
```

This example makes use of a Free procedure that explicitly frees the heap space used by the heap object pointed to by its parameter. Notice here how the Final procedure is used to ensure that all heap objects associated with the stack have been freed.

1.2.2 Kinds of Abstraction

This section discusses a useful classification scheme that can be used to decide among several package forms that can be used to structure data. The classification is based on the state information associated with the objects that can be derived from the package.

1. Subprogram libraries - Here the package contains only subprograms but not object or type declarations. Such libraries have no associated state.

2. Abstract objects - Here the package can contain both objects and subprograms but not types. The package can itself can be thought of as an "object" whose state is represented by the values of the objects it contains. As is the case for abstract types the representation of the objects within the package can be hidden. An abstract object is a good alternative to an abstract type when only one object with the type is ever created in the programs. This will frequently happen when large single instance tables are used. The abstract object is normally more efficient than an abstract type since the "object" is contained entirely within the package and need not be passed to each subprogram as a parameter.

3. Abstract types - Here the package can contain types and subprograms but not objects. Multiple abstract objects are created by using the visible type in multiple object declarations outside the package. The previous Stack examples illustrate this case. Another approach to multiple abstract objects is to instantiate a generic abstract object package multiple times. This approach is not recommended since the resulting object instances do not have a type and therefore can not be passed as parameters.

4. Related abstract types - When there are two (or more) related abstract types it is often useful to declare them within a single package. This allows both representations to be hidden from users, but at the same time routines in the package with parameters of both types can have access in their implementation to the representation of both types.

It is also sometimes useful to use some combination of the above techniques. For example, a package that contains objects, types, and subprograms is used in programs that need to mix the abstract object technique with the abstract type technique.

1.2.3 Derived Types

Derived types permit a new type to be derived from an existing type. The new type "inherits" the operations of the existing type. As a general rule derived types are not needed and should not be used.

There is however one place where the use of derived types is necessary. A type declaration of the form

```
type Int is range -1024..1024;
```

is actually equivalent to

```
type IntT is built_in_integer;
subtype Int is IntT range -1024..1024;
```

where IntT is an anonymous type and built_in_integer is one of the implementation defined integer types that is large enough to hold values in the range -1024..1024. Although the particular type selected is implementation dependent, the resulting subtype is implementation *independent*. This implementation independence is the reason why the use of derived types is necessary for all numeric types.

The only alternative to the use of derived types, is to use one of the built-in integer types. This must be avoided if transportability is desired, since the maximum range of these types can vary from one target machine to another (or even within different implementations for a single target machine). As an example of what to avoid, the type integer should never be used in a program.

```
type R is record
      X,Y: integer;    -- Bad programming style
   end record;
```

A more subtle use of the integer type appears in

```
type A is array (1..10) of boolean; -- Bad programming style
```

Here the index type of the array will default to integer and the program will be machine dependent. This is the reason why array index types should always be user declared.

There is a strong temptation to use the integer type for those integer variables where no range is obvious. A better approach is to declare

```
type my_integer is range -2**15..2**15-1;
```

The range should be selected to be sufficiently large to hold the largest expected integer value but also small enough to be supported by all Ada implementations of interest. Unlike programs that use the integer type, programs that use the my_integer type will be machine-independent.

1.3 Type Composition

This section discusses two techniques, type parameters and generic types, that can be used to generalize a type declaration so that parameters can be used to produce multiple related types at compile time and to produce multiple related subtypes at run time.

1.3.1 Type Parameters

Type parameters are a technique that can be used to separate the declaration of a type from the declaration of its subtypes. The Ada language provides several facilities which when used correctly will, in some cases, give the effect of parameterization of subtype constraints through an arbitrary number of structural levels. The following example illustrates each of these facilities.

```
type I10 is range -10..10;

subtype PI10 is I10 range 0..10;

type R(I:PI10) is record
      case I is
         when 0..5 =>
            X: I10 range 0..5;
         when 5..10 =>
            Y: I10 range 5..10;
      end case;
   end record;

type A is access R;

type V(J:PI10) is record
      Y: array (PI10 range 0..J) of A(J);
   end record;

type R1(K:PI10) is record
      T,U: V(K);
   end record;

Val: PI10;
V1: R1(5);
V1: R1(Val);
```

There are four basic techniques that are used to achieve parameterization here:

1. Range and accuracy constraints - These constraints serve as the parameterization technique for scalar types. It is not possible to control these constraints through multiple structural levels. For example when an integer subtype appears as a component of a record, then the range of this subtype must be specified when the record is declared and can not depend upon the discriminant parameters of the record.

2. Discriminants - This parameterization technique can only be used for records. In this case the formal parameters appear explicitly, as a discriminant of the record.

3. Constraint inheritance - This parameterization technique is used for access types. Any parameters of the underlying type are "inherited" by the access type.

4. Anonymous arrays used with discriminants - This technique is used for arrays. Here the array is embedded as the *only* component of a record with a discriminant. In this case the array type is anonymous; however, this causes no problems since the enclosing record type can be used as the effective type for the array. In the above example the user should use the object V1.T which has type V rather than the object V1.T.Y which has an anonymous type to refer to the entire array. When this technique is used, slices of the array will have an anonymous type and should be avoided. Note that index constraints can be used instead when parameterization of the array element type is not needed (in this case slices will not cause problems).

One further problem with parameterization should be noted: parameters must be used directly; that is, they may not be involved in computations. For example the following illegal example

```
type V1(J:PI10) is record
      Y: array (PI10 range -J..J) of boolean;  -- the -J is illegal
   end record;

X: V1(10);
```

would have to be handled instead by

```
type V1(JL,JH:PI10) is record
      Y: array (PI10 range JL..JH) of boolean;
   end record;

X: V1(-10,10);
```

These techniques can be used to generalize the previous stack example by parameterizing the stack size. This is done by deleting the declaration of StackSize and by replacing the private part by

```
type ElemArray(StackMax:StackSizeT) is record
      A: array(StackSizeT range 1..StackMax) of ElemT;
   end record;
type Stack(StackMax:StackSizeT) is record
      Top: StackSizeT range 0..StackMax;
      Elems: ElemArray(StackMax);
   end record;
```

Also note that .A qualifications must be inserted at the appropriate places in the package body.

1.3.2 Generic Types

Another way in which the Stack type can be generalized is by parameterization of the element type. Since types must be known at compile time, type parameterization is done by a compile time facility, generics. The stack can be made generic in its component type by deleting the ElemT type declaration from the visible part and then appending a generic part with an ElemT type parameter to the beginning of the package specification.

```
generic

   type ElemT is private;

package Stacks is
   ...  --same as before, except ElemT declaration has been deleted
end Stacks;
```

Instances of Stack are now created by generic instantiation.

```
package IntStacks is new Stacks(Int);
package BoolStacks is new Stacks(boolean);

SI: IntStacks.Stack;
BI: BoolStacks.Stack;
```

The previous approach to generic stacks has a problem: it is impossible to create a stack whose components are stacks. This is because Stack is a limited type while ElemT may not be limited. A second problem is that Final for the elements which are stacks will not be called at the needed times. This can be corrected by adding with clauses to the generic part.

```
generic
   type ElemT is private;
   with function Assign(LHS: out ElemT,RHS: in ElemT) is <>;
   with Final(E: in out ElemT) is <>;

package Stacks is
   ... -- same as before except an Assign routine is added for stacks.
end Stacks;
```

In addition to adding a definition for the Assign procedure, the package body will also need to be changed to use Assign instead of := for element assignment and to call Final just before the end of an element object lifetime. Since most types do not have an Assign or Final subprogram it will first be necessary to define them before a stack can be declared.

```
procedure Assign(LHS: out Int,RHS: in Int)
begin
   LHS := RHS;
end Assign;
procedure Final(E: in out Int)
begin
   null;
end Final;

package IntStacks is new Stacks(Int);
package IntStackStacks is new Stack(IntStacks.Stack);

SSI : IntStackStacks.Stack;
```

TUTORIAL ON ADA TASKING

Volume I: Basic Interprocess Communication

by

Stephen A. Schuman

31 March 1981

35

TABLE OF CONTENTS
(Volume 1)

1. Introduction and Overview

This is the first in a continuing series of volumes, the purpose of which is to present a tutorial introduction to the so-called "tasking facilities" embodied in the Ada programming language. Such a presentation can in no way serve as a substitute for information provided by the Language Reference Manual [LRM: Reference Manual for the Ada Programming Language, Proposed Standard Document, United States Department of Defense, July 1980]. We assume a thoroughgoing knowledge of that document on the part of every reader, and we also highly recommend a reading of relevant portions in the "Rationale" document which accompanied the preliminary definition of Ada [SIGPLAN Notices, Vol.14,6,B, June 1979], even though some of the material contained therein is now partially outdated.

The intended audience for this series consists of program (or system) designers, whence we are not primarily concerned with conveying the purely mechanical aspects of "coding" in (yet another) new programming language. Rather, our long-term goal is to develop a workable set of guidelines, whereby the program designer can naturally and effectively make use of the facilities in Ada to construct the kinds of complex softwwere systems which were meant to be supported by this particular language. Our subject area and assumed readership implies a high degree of familiarity with multi-process applications from the outset; the ideal reader will have already "trued out" the tasking facilities of Ada (at least on paper), an experience which may well have produced a certain sense of frustration. Such a reaction might be attributed in part to the fact that Ada does not seem to directly provide a practitioner with the traditional "tools of his trade" -- e.g., buffered message-passing or dynamic process-creation. Moreover, though primitives for each of these (or other) styles can theoretically be constructed within the Ada framework, this is not necessarily the most appropriate way to exploit the possibilities of the basic model embodied in this language.

For the foregoing reasons, we have opted to proceed from first principles in this series. The present volume introduces the essential concepts of the application domain under consideration, and covers the basic notions of interprocess communication in concurrent systems. Its sequel is devoted to higher-level structuring approaches within the Ada framework. Subsequent volumes will consist of more concrete examples, developed in the form a case studies.

2. By Way of Background

This section introduces the basic concepts with which we shall be concerned throughout this presentation. In particular, it will serve to characterize the category of programs that is of primary interest here, namely so-called "concurrent systems." The section concludes with a brief overview of the facilities in Ada for programming such systems.

2.1. Sequential Programs

Conventional computer programs, however large or complex, are normally formulated as a single sequential process. That is to say that they are defined in terms of a certain set of data objects, to which they are assumed to have exclusive access, together with a series of statements specifying what operations are to be carried out upon those objects. The statements in question will be executed one after another and, within each successive statement, its constituent expressions will also be evaluated in some pre-established order. Although such programs typically involve explicit transfers of control, occasioned both by conditional and iterative statements as well as by invocation of procedures or functions, and even though these constructs can generally be nested to an arbitrary depth, there is still only one "locus of control" associated with any execution of the program as a whole (whence it may be considered as a single process). This simple sequential model is sufficient to accommodate an extremely wide variety of computational tasks, including not only traditional data processing and scientific applications but also much of the system software required to support such programs (e.g., a compiler).

2.2. Concurrent Systems

There is however another very important category of programs, more properly referred to as "systems," which are most naturally formulated in terms of multiple sequential processes. Each such process embodies its own set of objects and separate series of statements (like a self-contained program), whence it is specialized to one particular role; the larger objectives of the overall system are then achieved by means of suitable intercommunication amongst these active entities. The archetype for this structural approach is of course to be found in social organizations, where a number of otherwise autonomous agents often act in collaboration to accomplish some collective function.

The underlying computational model for such systems involves concurrent execution of their constituent processes. Unlike purely sequential programs, a concurrent system has several distinct threads of control (one per active process), all of which evolve independently and (conceptually) in parallel. The only need for synchronization of their execution arises at points where two or more process must directly interact with one another. Otherwise, each may proceed at its own pace.

From the outset, it must be emphasized that the parallelism which is intrinsic to concurrent systems may well be more apparent then real, since the logical processes of any given application still have to be mapped onto a fixed number of physical processors. This mapping is usually achieved by imposing some appropriate scheduling discipline, whereby the actual execution of those processes will be at least partially serialized (and totally so in the case of a true mono-processor target configuration). Thus, the potential parallelism of this model does not necessarily represent a source of improved throughput. This latter goal can be realized only to the extent that it is possible to exploit whatever physical concurrency might be present in the underlying hardware, for instance by dedicating completely autonomous processors to specific active processes.

The real interest of the concurrent system model lies rather on a more fundamental level. In particular, this conceptual framework allows a designer to isolate possibly simultaneous or inherently asynchronous activities within a complex system, and then to formulate each one as a separate sequential process without having to specify in advance exactly how their execution is to be dynamically interleaved.

This latter property turns out to be essential for an increasingly important class of applications, encompassing conventional operating systems as well as real-time control systems, multi-station transaction systems etc. In attempting to characterize the application domain for which the Ada language was conceived, this class has been referred to as "embedded computer systems."

Perhaps the primary characteristic which distinguishes such applications is that they are all event-driven systems. In essence, their sole reason for existence is to respond to a variety of external stimuli -- ranging from operator and end-user requests to initiate or terminate some specified (high-level) transaction down to device interrupts and timing signals generated by the (more or less specialized) hardware they serve to control. Much of the intrinsic complexity of these applications arises from the fact that the events in question occur asynchronously: neither their actual order of arrival nor their relative ordering can be prescribed in advance (since they are, in effect, a manifestation of the "outside world").

Obviously, the software which serves to control such a system lies at the opposite end of the spectrum from traditional data processing or scientific applications, wherein all external transactions (e.g., file accesses) are -- or at least appear to be -- fully synchrononous operations, invoked by the execution of a single sequential program. The asynchronous (and thus naturally concurrent) character of the external events that ultimately drive the class of systems considered here imposes from the beginning a logical organization which comprises multiple independent processes.

Very schematically speaking, there will exist a separate process to handle each (physical or logical) "resource" within the system, along with additional processes to carry out the various different "activities" which could conceivably be in progress at any given time. Moreover, this same basic structuring principle can (and generally will) be re-iterated at successively higher "levels of abstraction," corresponding approximately to autonomous subsystems which perform progressively more complex functions within that overall organization. The requisite internal synchronization, whereby proper operation of the system as a whole is established and maintained, must then be embodied in the "protocols" which govern the possible interactions among this multiplicity of processes and subsystems.

2.3. Synchronous Communication

The purpose of synchronization within any concurrent system is to impose the minimum constraints upon the (otherwise unconstrained) order in which each independent process may legitimately proceed, such that the larger objectives of the system as a whole will always be achieved. As stated initially, the need for synchronization only arises when two or more processes must interact with one another, whence it is intrinsically a matter of interprocess communication (i.e., involving at least two distinct parties). The "such that" proviso put forth just above implies, in the final analysis, that each constituent process of the system will always perform its own role in a "socially acceptable" fashion. This comes down to requiring that every separate process be able to maintain the integrity and consistency of its own internal state. Only under these conditions can it guarantee that any interactions with other processes will be "meaningful" (at least from its standpoint); if all parties to every transaction are able to make the same guarantee, however, then it must be true that the overall system will operate as expected regardless of how execution of the constituent processes is actually interleaved in time. (This argument presumes, of course, that these processes and their pattern of intercommunication were properly specified in the first place -- wherein lies the real challenge of system design!)

The essence of synchronization is then to delay any act of interprocess communication until such time as all of the participants directly affected by that particular transaction are prepared to change their internal states in a mutually consistent fashion. (It is precisely this delay which provides the occasion for some form of scheduling to intervene, whereupon a choice may be made amongst pending transactions for which all parties involved are either ready to proceed or have already completed their interaction.)

40

Aside from delays explicitly introduced for synchronization (however this is accomplished), the semantics of concurrent computation says that the active processes within a given system should always be considered as if they were in fact executed in parallel, albeit at an undefined rate of progress relative to one another. The reason for following this precept is to avoid all implicit assumptions regarding the specific scheduling strategy, the physical characteristics of the underlying hardware or the actual order in which external events arrive. Any such assumptions could lead to time-dependent anomalies under even slightly altered conditions (and therefore represent a source of design errors that are especially difficult to detect -- and often almost impossible to correct "after the fact").

It is generally well understood that this basic design principle effectively precludes unsynchronized accesses to global data -- i.e., "shared variables" -- as a safe means of interprocess communication. If two separate processes are indeed executed in parallel, one of which updates a certain data structure while the other reads it (which presumably will occur if they are trying to communicate in that manner), then those data accesses must be regarded as asynchronous and therefore unsafe (since the respective references and assignments could well be interleaved in a wholly arbitrary and not necessarily atomic fashion). The need to synchronize access to such data implies that there are only two approaches which are always safe: either those variables must be made local to one process or the other (in which case they are no longer global, but rather part of the internal state of a specific process); or, alternatively, they must be confided to a third party which will act to ensure the requisite synchronization (whereupon they become local to this intermediary process, which then plays the role of a "shared resource" that serves to coordinate communication between the two original processes). Whichever alternative is adopted, the net effect is to reduce all interactions to synchronous communication -- which remains the only viable way for independent processes to interact.

Quite apart from whatever sequentialization might be introduced by implicit scheduling, the potential parallelism within a given system may well be highly constrained, or even precluded entirely, as a result of the particular protocol adopted for interprocess communication. If, for example, a data acquisition process is specified so as to await confirmation that each item produced has reached its ultimate consumer before starting to acquire another, then the system will in fact operate in a purely sequential fashion. On the other hand, if the process in question is allowed to proceed with the acquisition of another item immediately upon delivery of the previous one to some intermediary (perhaps the first among many), then there exists at least the possibility for parallel activity -- whether or not this can be supported by the actual hardware. Thus the specific pattern of synchronous communication amongst the various processes determines the degree of logical concurrency embodied in the design of that system. As a general rule, in the absence of other constraints one will seek to maximize logical concurrency.

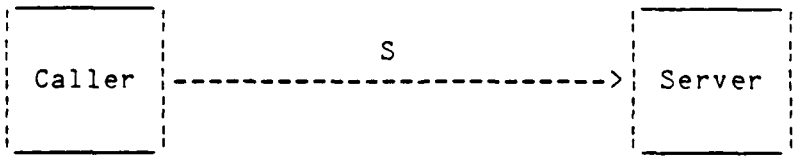
2.4. Concurrency in Ada

A sequential Ada program consists solely of subprograms (no matter how deeply nested or structurally "packaged"), and can therefore be executed as a single (externally initiated) process. In the more general case, however, a complete "main" program will consist of multiple sequential processes, each of which is represented by means of a separate task declaration, together with its associated task body (where this body serves to define the potentially parallel path of execution corresponding to an independent process). Thus an Ada task constitutes the unit of logical concurrency within that language. Any program comprising one or more task declarations may therefore be construed as a concurrent system.

Reflecting the fundamental requirements for such systems, the synchronization primitives embodied in Ada are based on explicit interprocess communication -- an approach inspired primarily by the "Communicating Sequential Processes" model originally introduced by Hoare [CACM, Vol.21,8 August 1978]. Moreover, Ada supports this synchronous communication in the form of an essentially procedural interface between exactly two (necessarily distinct) sequential processes:

- the caller task, which initiates a new transaction by requesting some particular named service (say S);
- the server task, which responds to each such request by carrying out the operation associated with S (if any).

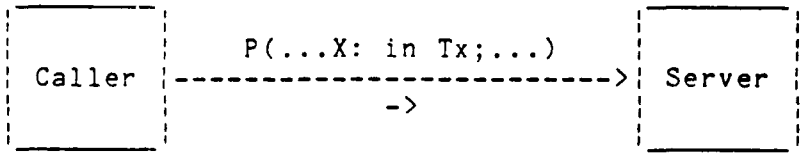
We shall schematically depict this form of communication as follows:



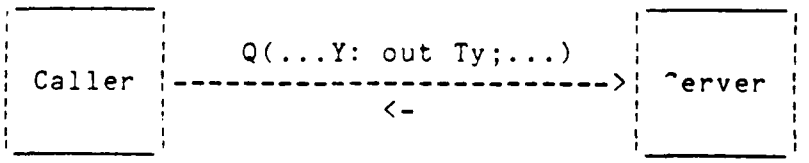
It may be observed that the caller/server relationship is inherently asymmetric, reflecting the flow-of-control between the two parties involved in every such transaction. The initiative lies entirely with the caller, who actively requests a specific service (which is identified by name as part of the call itself). The server plays a more passive role, simply responding to the individual requests as they occur (without even knowing the identity of the caller being served in each instance); it is nonetheless the server who establishes the meaning of such calls (as defined by the associated operation), and who thereby determines the point at which any given transaction is considered to be complete (whereupon the corresponding caller may once again proceed).

In the context of such transactions, it is also possible for information (as well as control) to flow between the two participant processes. The passage of data is achieved by means of formal parameters, one or more of which may be associated with the particular operation to be invoked. These are specified in accordance with the same conventions that apply for Ada procedures:

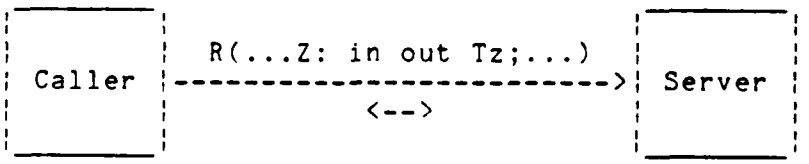
Transmission of Data:



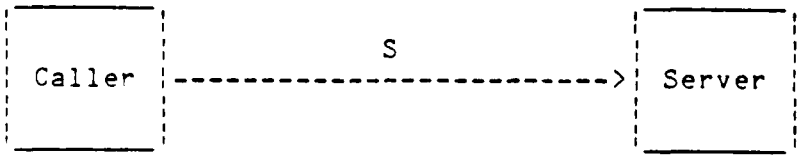
Reception of Data:



Interchange of Data:



No Passage of Data:



Hence both the extent the direction of data-flow are wholly independent of the basic flow of control for interprocess communication.

43

The underlying control relationship involved in this form of communication may be viewed as establishing a "connection" between the caller process and the server process, which lasts for the duration of each individual transaction. Information may or may not flow across such a connection, depending upon the associated parameter specification: if there are arguments to be passed, they will be transmitted from caller to server upon initiation of a given transaction; if there are results to be returned, they will be transmitted from server to caller upon subsequent completion of that transaction; if there are no formal parameters, then only the basic control signals (corresponding to making and breaking of the connection) need be exchanged between the two intercommunicating processes.

It can be seen that, from the standpoint of the calling process, the semantics of synchronous communication in Ada are exactly the same as for a conventional procedure call. This comes about because the caller task is effectively suspended until the corresponding transaction is complete, just as if it had invoked an ordinary procedure. (Furthermore, if for any reason a certain transaction cannot be completed successfully, this failure will be signaled by the raising of an exception within the context of the calling task, as for any unsuccessful subprogram call.)

It is on the side of the server, rather than of the caller, that additional steps must be taken in order to ensure the synchronization required for interprocess communication. A simple procedure call cannot safely be used for such communication any more than direct access to global variables, since that procedure is in effect executed as an extension of its caller (and hence conceptually in parallel with the corresponding server). Moreover, it may be the case that a given procedure is called from more than one process, whereupon there could be several separate instances of that procedure executing concurrently. This potential parallelism poses no particular problem concerning accesses to local variables and/or formal parameters of that operation, since these entities are (by definition) distinct for every invocation. If however the procedure in question accesses any global data (so as to communicate with the server), then all of those accesses must again be regarded as asynchronous.

For purposes of synchronous communication, Ada provides an alternative form of definition, wherein each such operation is declared to be an entry belonging to the server process. This is specified as part of the corresponding task declaration:

```
task Server is
   ...
   entry E(...);
   ...
end;
```

This distinction between defining a particular operation to be an entry, as opposed to a procedure, comes down to specifying that concurrent calls should be serviced sequentially instead of in parallel. For an entry, the meaning of the operation in question is established by one or more accept statements, effectively appearing in the imperative part of the associated task body (and, in almost all cases, embedded within some sort of loop):

```
task body Server is
   ...
begin
   ...
   loop
      ...
      accept E(...) do
         ...
      end;
      ...
   end loop;
   ...
end Server;
```

An accept statement serves to define the sequence of actions (those enclosed between the delimiters do ... end) to be executed for the next of the (possibly many) pending calls to the same named entry, during which time the server is said to be in rendezvous with the corresponding caller (so that the server has access to the parameters of that particular invocation as well as to its own data); only after the actions specified within such an accept statement have been carried out is that entire transaction considered to be complete, whereupon the caller and server processes may once again proceed concurrently. In the simplest case, when there are no actions to be performed during the rendezvous (nor any formal parameters), the accept statement may be reduced to its degenerate form:

```
accept S;
```

In such situations, the entry S may be regarded as a pure "signal," the next pending call to which is simply "acknowledged" by this statement.

Thus the acceptance of successive calls becomes a fully synchronous operation, embodied in an executable statement (unlike a procedure, the meaning of which is defined by a purely declarative construct). Accordingly, the server itself can control the points at which communication with its caller(s) may occur. Within this framework, concurrent calls are conceptually enqueued on the designated entry (in their actual order of arrival), from where they will be served one after another. If there are no calls outstanding for an entry that is ready to be accepted, then the server process is delayed on its side until such time as that accept statement can be completed.

45

In addition to a simple accept statement, Ada also allows the programming of a so-called selective wait:

```
select
   accept E1(...) do
      ...
   end;
   ...
or
   accept E2(...) do
      ...
   end;
   ...
or
   ...
or
   accept En(...) do
      ...
   end;
   ...
end select;
```

This permits the server to make a (non-deterministic) choice amongst potentially pending calls to several different entries E1, E2, ... En, and then to perform some suitable sequence of actions according to which particular transaction was selected. Moreover, the accept statements appearing in a selective wait may be individually guarded by means of a boolean expression, which must be true in order for the associated alternative to be selected:

```
select
   when G1 =>
   accept E1(...) do
      ...
   end;
   ...
or
   ...
or
   when Gm =>
   accept Em(...) do
      ...
   end;
   ...
end select;
```

By appropriately setting the values of local variables figuring within the guards G1...Gm, the server task may exercise finer control over which entries are actually "open" each time this selective wait statement is executed.

In terms of these facilities, the server process can be explicitly programmed to maintain the consistency of its own data and, thereby, to ensure the correctness of operations by which it communicates with other processes -- irrespective of the number of calls that might be pending concurrently (which necessarily emanate from distinct callers). Hence, it is the server which ultimately establishes the synchronization required for interprocess communication. Any given caller merely initiates a request for some particular transaction and then waits for that transaction to be completed, exactly as in the case of an ordinary procedure call.

Thus, there is no need for the calling processes to be in any way concerned with whatever additional synchronization might be imposed by the server for its own purposes. As such, it is sufficient for the server to present a purely procedural interface to its potential callers. Within this framework, the design of a concurrent system therefore proceeds by methods which are, in many respects, analogous to the decomposition of a sequential program into a collection of subprograms which invoke one another. In particular, the key to specifying an appropriate pattern of intercommunication amongst the constituent processes of such a system (and thus, to establishing the existence of distinct processes and their associated interfaces during successive stages of decomposition) is to define, in first instance, the desired caller/server relationship within a proposed configuration. This initial step may be carried out in exactly the same way as for a sequential system -- i.e., as if the transactions in question were conventional procedure calls; all further considerations, insofar as they relate only to synchronization issues, can then be addressed solely in the context of the corresponding server definitions.

An Ada task may have zero, one or more entries defined as part of its interface; similarly, the associated task body may nor may not call upon the operations provided by other tasks. The resultant caller/server relation is what establishes the overall pattern of interprocess communication within a given system. In general, a task representing some particular process may therefore be both caller and server. There are, however, two special cases which we shall categorize as follows:

- a <u>resource</u>, corresponding to a process which is purely a server;

- an <u>activity</u>, corresponding to a process which is only a caller;

Experience has shown that this (rather intuitive) taxonomy nonetheless provides a useful frame of reference, especially as one proceeds to design of larger-scale concurrent systems (a problem which will be taken up in the sequel to this volume).

3. Basic Interprocess Communication

By way of illustrating the basic principles of interprocess communication within the Ada framework, we shall build upon a very much simplified, albeit archetypical, example. At the conceptual level, the system in question consists of just three processes.

- a PRODUCER process, which is dedicated to data acquisition (e.g., input from some external source or device), where the data items so acquired are delivered one at a time as output from this process.

- a TRANSDUCER process, which serves to carry out some (potentially quite complex) transformation on the data supplied as input, where the corresponding result for each such argument is delivered as output.

- a CONSUMER process, which is responsible for the final disposition of the result data (e.g., output to some display or storage medium), where the successive data items are supplied as input to this process.

The intended flow of information amongst these three processes is shown pictorially in Fig. 3-1. In terms of data-flow, it can be seen that the constituent processes of this system are organized into a conventional "pipeline" (or "bucket brigade") configuration.

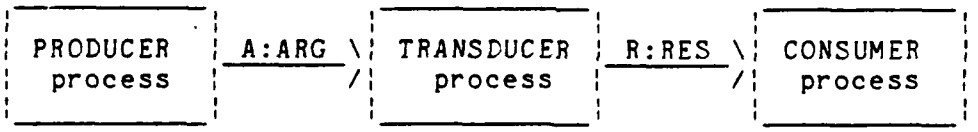


Figure 3-1: Data-Flow of the Example Multi-Process System.

3.1. Initial Formulation in Ada

Turning to the formulation of this example in Ada, the simplest possible program structure will be adopted at the outset. More realistic approaches to the overall structuring of such systems are to be addressed in the next volume of this series. For the moment, however, it will be assumed that the particular application of interest can be conveniently expressed as a single "main program."

In skeleton form, the definition of such a main program might appear as follows:

```
procedure Application is

   type ARG is array ... of ... ;
   type RES is array ... of ... ;

   task Producer is ... ;

   task Transducer is ... ;

   task Consumer is ... ;

   task body Producer is
      ...
   begin
      ...
   end Producer;

   task body Transducer is
      ...
   begin
      ...
   end Transducer;

   task body Consumer is
      ...
   begin
      ...
   end Consumer;

begin
   null;
end Application;
```

Formulated in this manner, the constituent processes of the system are specified directly in terms of declarations for the corresponding tasks (Producer, Transducer and Consumer), together with their associated bodies. As such, the entire system is effectively defined within tne declarative part of the enclosing main program (Application), whereupon the statement list of this latter is empty. Thus, a given execution of Application (as an externally initiated task) consists simply of activating its three component tasks, which then proceed to communicate amongst themselves; that execution is finished only when all of those dependent tasks have properly terminated, owing to the implicit "join" at the end of the embedding (main) task.

3.2. Simple Patterns of Communication

Once such an overall system structure has been established, the primary question then becomes how to specify the requisite pattern of communication among its constituent processes. The intended flow of information has already been fixed (cf. Fig. 3-1), namely:

- items of type ARG are to be transmitted from the Producer to the Transducer process;

- items of type RES are to be transmitted from the Transducer to the Consumer process.

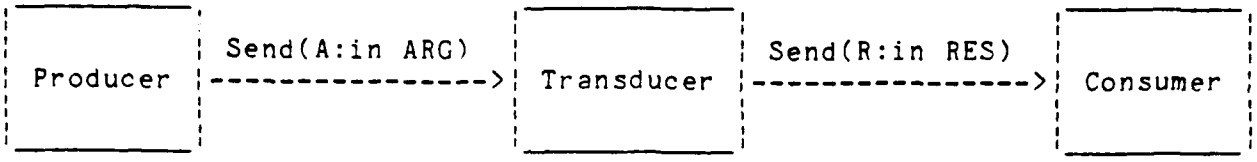
Having chosen here to explicitly represent each of these processes as an Ada task, it follows that this intercommunication must be carried out in a fully synchronous fashion, making use of the entry call/accept mechanism. Indeed, no other approach would be viable, since these language facilities embody the only safe means whereby individual tasks may directly interact with one another.

Even though the possible approaches to interprocess communication are so constrained, the program designer is nevertheless left with a very important degree of freedom in this regard. Specifically, the client/server relationship (i.e., "who should call whom") can be decided upon independently of the desired data-flow. Thus, for both of the elementary transactions described above, the flow of control need not necessarily correspond to the direction in which information is to flow. In the present context, therefore, four distinct patterns of communication might well be considered. These alternatives are depicted schematically in Fig. 3-2, on the next page.

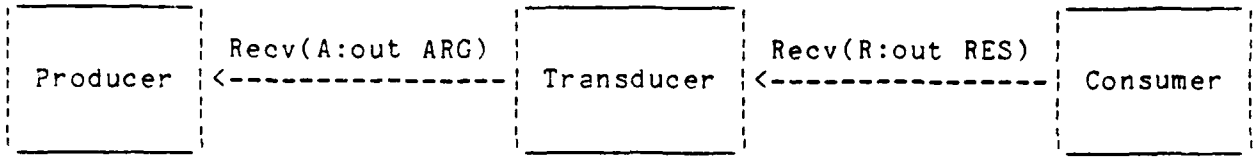
All of the communication patterns shown in Fig. 3-2 are equally plausible, although there are advantages and drawbacks associated with each approach. The choice ultimately depends upon the expected behavior of the system as a whole. For this reason, we shall examine these four alternatives one after another in the subsections which follow, attempting to provide some insight into the basis upon which such decisions are made. It will emerge from the ensuing discussion that these considerations play a quite critical role in the program design process.

Asymmetric Communication Patterns

Forward-Directed:

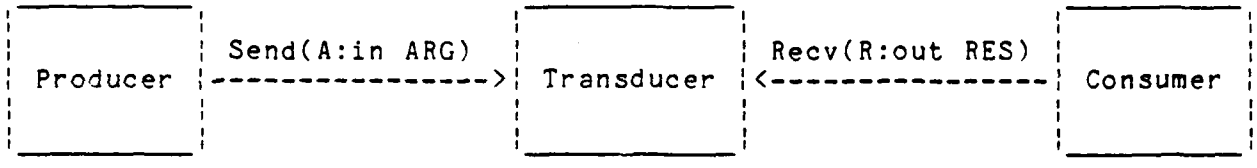


Backward-Directed:



Symmetric Communication Patterns

Inward-Directed:



Outward-Directed:

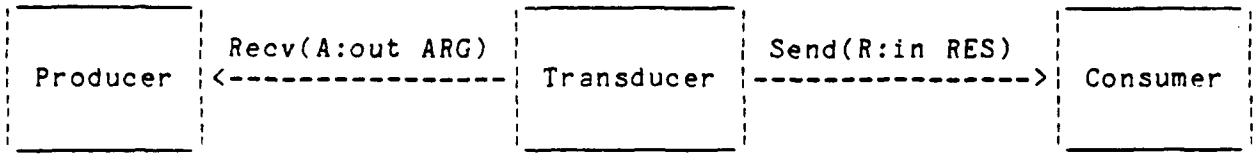
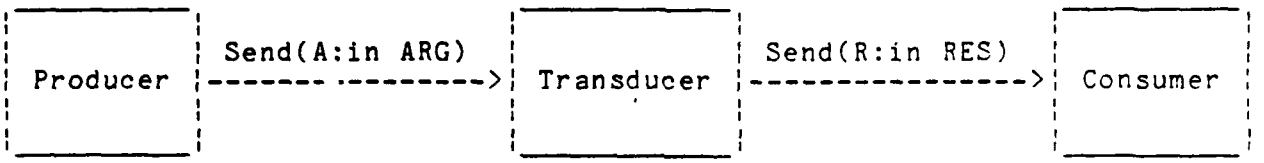


Figure 3-2: Alternative Patterns of Communication in the Example System.

51

3.2.1. Forward-Directed Communication

Pursuing the development of our example system, we shall opt initially to structure the interrpocess communication as follows:



Such a pattern of communication is referred to as "forward-directed," because the flow of control goes in the same direction as the flow of information. On purely intuitive grounds, this might seem to be the most natural approach. For this reason alone, it has been adopted here as a point of departure; as yet however, there is no pragmatic basis for preferring this particular approach over any of the other alternatives depicted in Fig. 3-2.

It is perhaps useful to characterize the configuration shown above according to the terminology introduced earlier. At this level of description, the Producer process represents an activity (since it is merely a caller), whereas the Consumer process constitutes a resource (since it is only a server). The Transducer process is neither activity nor resource (being both server and caller), which reflects its intermediary role in this organizational structure.

Given the communication pattern specified in the above diagram, a corresponding Ada formulation can be obtained by appropriately refining the overall structure outlined previously. Proceeding in this fashion, the constituent processes of the system would now be represented (within the declarative part of the main program Application) by the following (complete) task declarations:

```
task Producer;

task Transducer is
   entry Send(A:in ARG);
end;

task Consumer is
   entry Send(R:in RES);
end;
```

The associated bodies would then be defined (still in skeleton form) as follows:

```
task body Producer is
   ...
begin
   ...
   Transducer.Send(...);
   ...
end Producer;

task body Transducer is
   ...
begin
   ...
   accept Send(A:in ARG) do
   ...
   Consumer.Send(...);
   ...
end Transducer;

task body Consumer is
   ...
begin
   ...
   accept Send(R:in RES) do
   ...
end Consumer;
```

Although these task bodies are not yet complete, this structure is now sufficient to exhibit the pattern of interprocess communication currently under consideration:

- the Producer calls Transducer's entry Send which, when accepted, allows transmission of an item of type ARG;
- the Transducer calls Consumer's entry Send which, when accepted, allows transmission of an item of type RES.

This framework therefore provides a basis for sketching out a possible formulation for the "steady-state" operation of the entire system (i.e., disregarding for the moment the problems of process initiation and termination).

3.2.1.1. Steady-State Operation

A first definition for the steady-state operation of the example system could be obtained by completing its task bodies as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      Transducer.Send(AA);
   end loop;
end Producer;


task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      accept Send(A:in ARG) do
         AA := A;
      end;
      Transform(AA,RR);
      Consumer.Send(RR);
   end loop;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
begin
   loop
      accept Send(R:in RES) do
         RR := R;
      end;
      Dispose(RR);
   end loop;
end Consumer;
```

Specific details of the data acquisition, transformation and disposition operations are of no immediate interest here, whence the corresponding actions have been encapsulated as local procedures, which will not be further defined.

What is of definite interest, however, is the flow of information between the constituent processes of our example system. For the pattern of communication considered at present, the actual data transfers must be defined by means of appropriate accept statements within the bodies of the Transducer and Consumer tasks, respectively. Recall that an accept statement specifies a sequence of actions to be carried out whilst the caller and server tasks are in "rendezvous." In each instance above, those actions have been reduced to a single assignment statement, whereby the input item supplied by the caller is explicitly copied into a local variable belonging to the server. Once this has been accomplished, both processes may again proceed independently. Thus, the objective has been (as it will continue to be throughout this presentation) to maximize the degree of logical concurrency within the system as a whole.

On grounds of efficiency, the copying necessarily associated with this transmission of information between processes might at first seem to be a cause for concern (especially since the types ARG and RES were declared as arrays, suggesting that such items could be quite large blocks of data). If these copies are to be avoided, however, then the entire application may as well be formulated as a single (main) task:

```
procedure Application is

   type ARG is array ... of ... ;
   type RES is array ... of ... ;

   AA: ARG;
   RR: RES;

   procedure Acquire(A:out ARG) is ... ;
   procedure Transform(A:in ARG; R:out RES) is ... ;
   procedure Dispose(R:in RES) is ... ;

begin
   loop
      Acquire(AA);
      Transform(AA,RR);
      Dispose(RR);
   end loop;
end Application;
```

Whereas this definition still shows only the steady-state operation, it can be seen that the system has now become totally sequential; acquistion of a new item does not begin until the transformation, and indeed the disposition, of the previous one has been completed. With such an approach, no concurrency is present (or possible at this level).

3.2.1.2. Startup and Shutdown

Returning to the multi-task formulation of our example, we are now ready to address the issues of process initiation and termination, which provide the basis for defining startup and shutdown of the overall system.

From a careful reading of the Ada Reference Manual [9.3], it may be ascertained that the first steady-state definition of this system, as developed above, presents no particular problems insofar as startup is concerned. Its constituent processes, specified in the context of the main program Application, are initiated implicitly as a consequence of the corresponding task declarations, whereupon each of these tasks (as defined by their associated bodies) then proceeds as a parallel path of execution. Although the language does not specify an order in which such tasks are activated, this turns out not to matter so long as their interactions are entirely confined to synchronous communication (via the entry call/accept mechanism). When this rule is respected, it will in general be true that system startup requires no special attention, since process initiation essentially operates in accordance with the underlying block-structure of the language.

Unfortunately, the same principle of "benign neglect" does not apply with respect to system shutdown. The semantics of Ada are such that execution of a statement list appearing within most block-like constructions (including both subprogram and task bodies, but excluding the optional initialization part of a package body) is not considered complete until all locally-declared (dependent) tasks have terminated [cf. LRM 9.4]; in other words, there is an implicit join operation associated with exiting from any of these language constructs. Thus, even though process termination like initiation is based on the underlying block-structure, the program designer must pay explicit attention to ensuring that the component tasks at every level terminate in one fashion or another, lest the entire system become suspended at some point waiting for completion of a subtask which (inadvertently) continues forever. For this reason, the problems of process termination must always be addressed as an integral part of program design. It will be seen, moreover, that these considerations often play a quite critical role in the overall structuring of any given system.

Examining the previously developed definition for the steady-state operation of our example, it can be seen that the system as formulated so far will indeed never shut down; this is hardly surprising, since each of the task bodies for its constituent processes was initially expressed in terms of an infinite loop.

When it comes to deciding the basis for effecting shutdown in the context of our example, we are in fact facing one of the more difficult problems involved in system (as opposed to program) design. It is well known that the issues which arise in conjunction with both the startup and shutdown of any concurrent sytem are often far more complex than those which pertain to its steady-state operation. For this reason, it will in general be necessary to consider a number of alternative strategies in this area before settling upon one particular approach. We shall, however, defer the discussion of such alternatives to a later section and adopt, for the moment, what is presumably the most intuitive strategy, namely to shut the system down once the Producer process has determined (by some purely local criterion) that the last item of input has been transmitted; we therefore refer to this mode of operation as "input-driven." The constraint we wish to impose in this connection, however, is that the shutdown of our system will be "graceful" -- i.e., that all of its constituent processes will terminate properly, but only after the result corresponding to the last input argument has been delivered by the Transducer process and this result has been duly disposed of by the Consumer process.

To simplify the presentation, we shall express the decision as to whether the input has been exhausted in terms of a local loop counter, whereupon the body of the Producer task may be reformulated as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   for N in range ... loop
      Acquire(AA);
      Transducer.Send(AA);
   end loop;
end Producer;
```

Thus, the Producer process terminates normally after some specified number of arguments have been sent to the Transducer. The problem then becomes one of ensuring the proper termination of the processes which are "downstream," once they have also completed action on the last item of information in the pipeline. We reject here the trivial solution of merely introducing a separate local counter (with the same range) into the bodies of the Transducer and Consumer tasks as well, since this would be tantamount to sharing "global" knowledge amongst all of the constituent processes (thereby violating our requirement that shutdown should be based on a determination made solely by the Producer). We are therfore obliged to resolve this problem by other means, necessarily involving more explicit forms of synchronization.

At this stage we have now provided for explicit termination of the Producer task (by means of a simple loop counter) but, for the moment, left both the Transducer and Consumer tasks as first formulated (in terms of an infinite loop). It is instructive to consider here exactly why the system so defined fails to shut down as desired. With regard to the flow of information, it can be seen that the Transducer and then the Consumer will indeed successfully complete action upon the last data item delivered by the Producer. However, both the downstream processes will subsequently block, awaiting transmission of the next item to be processed; each will wait forever, because there are no further arguments to come from the Producer (which has terminated) nor, therefore, will any more results be transmitted from the Transducer to the Consumer. Thus, neither of the latter processes will ever terminate (since the corresponding tasks will simply wait indefinitely on their respective accept statements). In consequence the overall system will remain deadlocked in this waiting state (i.e., execution of the main program Application will never be complete, owing to the fact that some of its component tasks do not terminate).

The problem encountered at this point is merely another manifestation of the issue that always arises in conjunction with pipeline systems like the present example -- namely, how to signal (or otherwise detect) an end-of-transmission, such that the processes waiting downstream do not remain forever blocked in that state. This problem may be resolved in several different ways.

A very common approach is to transmit some distinguishable end-of-stream marker in place of (or in addition to) the expected data item. In the current context, this might for instance be accomplished either by declaring each of the types ARG and RES to be a variant record structure or, alternatively, by specifying some form of status indicator as a second parameter to the Send entry of both the Transducer and Consumer tasks. We shall leave the corresponding Ada formulations as an exercise for the reader, since neither is particularly difficult to program (though both lead to certain language complexities and/or run-time overhead in their own right). We note, however, that all such "data-directed" approaches involve passing additional information as part of every transaction between communicating processes, whereby the encoding conventions also serve to ensure the requisite synchronization (e.g., to trigger shutdown). This may be acceptable (or necessary) in many situations, but our interest here is rather to illustrate how the same effect may be achieved more directly, by means of appropriate control signals.

The potential for deadlock as described above, where a substask ends up waiting indefinitely on an accept statement when it should simply terminate normally, is so prevalent that Ada provides a special language feature to aid in programming orderly shutdown under these conditions. The basic principle is to replace every such accept statement by a selective wait statement having a "terminate" branch. This latter alternative (which may also appear in multiple selective waits) causes immediate termination of the associated task; it can be selected if and only if all other tasks in the same context are either already terminated or waiting on a similar open alternative [cf. LRM 9.7.1]. This permits the problem of shutting down our example system to resolved by so reformulating the Transducer and Consumer task bodies:

```
task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      select
         accept Send(A:in ARG) do
            AA := A;
         end;
         Transform(AA;RR);
         Consumer.Send(RR);
      or
         terminate;
      end select;
   end loop;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
begin
   loop
      select
         accept Send(R:in RES) do
            RR := R;
         end;
         Dispose(RR);
      or
         terminate;
      end select;
   end loop;
end Consumer;
```

When this specialized facility is employed, the synchronization required to achieve proper termination is provided by an implicit (language-defined) signal. Conceptually, a built-in condition is raised whenever all "active" processes in a given system have separately (and for whatever reasons) come to the end of their own execution; this event is only acknowledged, however, when every other "passive" process of that system has also entered a quiescent state (being ready either to accept further communication or, alternatively, to complete its execution as well), whereupon the entire system can be shut down in a fully synchronous fashion. In our example as now formulated, such a signal is effectively generated by termination of the Producer process (once it has successfully transmitted the last input argument to the Transducer), but termination of the overall system is deferred until both the Transducer and Consumer processes have reached their respective quiescent states. It may or may not be obvious that this latter property is sufficient to ensure that the last result will indeed be properly disposed of by the Consumer before shutdown. The argument, based on the rendezvous semantics of Ada, goes as follows:

1. The Producer cannot terminate until it has completed transmission of the last argument to the Transducer.

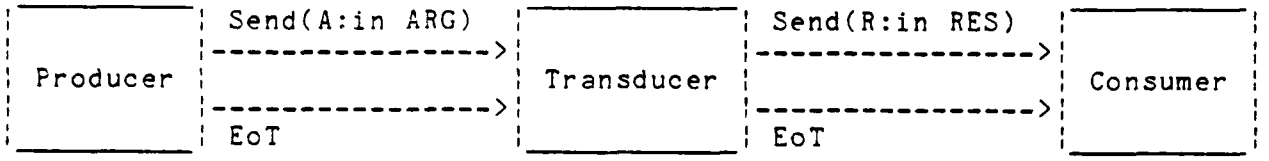
2. The Transducer cannot re-enter its selective wait until it has transmitted the corresponding result to the Consumer.

3. The Consumer cannot re-enter its selective wait until it has finally disposed of the last result from the Transducer.

Only after all three of these conditions have been satisfied can the system shut down -- whence it will (eventually) do so gracefully, as we have required.

Whereas the language mechanism employed above may at first strike one as something of a trick, it nonetheless provides not only further insight into the general problem of termination but also a ready-made solution, at least in certain simple cases. As such, it should always be given due consideration, even if it is finally rejected because of certain accompanying drawbacks. Among these latter is the fact that it forces the program designer to reason about all of the constituent processes at once in order to ensure shutdown of the system as a whole. It would be preferable, from a methodological viewpoint, to be able to think more in terms of the separate transactions between individual processes that directly communicate with each other (applying the principle of divide and conquer). Moreover, it may be observed that this approach to termination is ultimately based upon another form of centralized knowledge (embodied in the run-time support system for the language, which is capable of determining the state of every component task in the system); this is contrary to the overall objectives pursued here.

For the foregoing reasons, we now wish to explore an alternative approach to shutting down our example system, in order to show how this effect can also be achieved by means of explicit (programmer-defined) signals between communicating processes instead of relying upon implicit signals -- i.e., those which underlie the particular termination mechanism that has been built into the language. Specifically, this alternative consists of introducing separate end-of-transmission signals, so as to properly terminate the (logically distinct) transactions between the Producer and the Transducer, and between the Transducer and Consumer, respectively. Thus, such a signal will be seen to serve the same synchronization function as an end-of-stream marker in traditional data-directed approaches (with the potential advantage that the requisite information need only be transmitted once, at what is actually the end of the stream, since the desired conventions are effectively embodied in the communication protocol itself rather than being encoded in some concrete data representation). For the forward-directed pattern of communication currently under consideration, the additional control signals would be introduced as follows:



These signals are necessarily directed in the same way as the calls by which the associated data stream is transmitted, so that the initial characteristics of the overall configuration are in no way altered (the Producer is still an activity and the Consumer a resource, while the Transducer continues to play its intermediary role). The introduction of these control signals can be accommodated by simply adding the corresponding (parameterless) entries to the Transducer and Consumer tasks. The constituent processes of our system would then be defined by revising both the task declarations and their associated bodies:

```
task Producer;

task Transducer is
   entry Send(A:in ARG);
   entry EoT;
end;

task Consumer is
   entry Send(A:In ARG);
   entry EoT;
end;
```

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   for N in range ... loop
      Acquire(AA);
      Transducer.Send(AA);
   end loop;
   Transducer.EoT;
end Producer;


task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      select
         accept Send(A:in ARG) do
            AA := A;
         end;
         Transform(AA,RR);
         Consumer.Send(RR);
      or
         accept EoT; exit;
      end select;
   end loop;
   Consumer.EoT;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
begin
   loop
      select
         accept Send(R:in RES) do
            RR := R;
         end
         Dispose(RR);
      or
         accept EoT; exit;
      end select;
   end loop;
end Consumer;
```

For this alternative formulation, the property of graceful shutdown can be established by exactly the same arguments as for the previous approach (based on implicit signals). Furthermore, it may be seen that the basic structure of these two solutions is essentially identical. The only fundamental difference is that, in this second formulation above, the necessary synchronization is programmed explicitly: at the end of each separate data stream, the originating process transmits a mutually agreed upon control signal (here called EoT in both cases) to the receiver with which it has been communicating; the originator then awaits confirmation of this signal, after which it terminates normally of its own accord (as do all of the downstream processes as well, each in their own time).

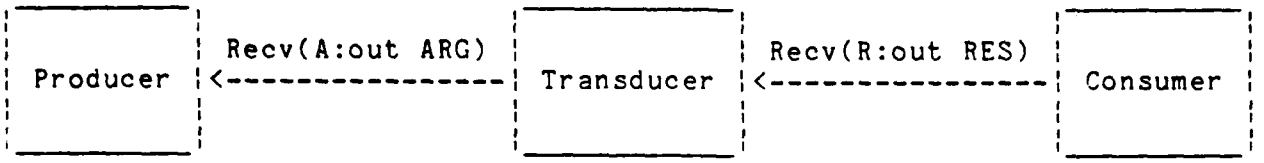
Thus, by contrast with the first solution, this explicit approach to shutting down the system might also be said to be more decentralized, in that each of the constituent processes takes local responsibility for properly terminating whatever transactions it has previously initiated, prior to completing its execution. (We note that such a discipline is entirely precluded by the built-in mechanism of Ada, since selection of a terminate alternative causes immediate termination of the task in question, leaving no opportunity for further communication or any other form of "finilization" activity).

It should be pointed out that both solutions developed above depend critically upon the fact that all communication between processes is carried out in a synchronous fashion (based primarily upon the rendezvous semantics of the language). With regard to actually transferring data from one independent process to another, the need for some such synchronization is presumably obvious. Our requirement for graceful shutdown of the system has made synchronizing the termination of its constituent processes equally essential (irrespective of whether this is accomplished explicitly or implicitly). Consequently, any attempt to terminate the downstream processes asynchronously -- for instance, by raising the FAILURE exception in the corresponding tasks -- would be wholly inappropriate (because it could not then be guaranteed that the items already in the pipeline were processed to completion); the same argument applies (with even greater force) to use of the abort statement.

A final lesson to be learned from the more general termination problem addressed here is that the program designer should always be exceedingly suspicious of an isolated accept statement, as opposed to a selective wait which makes some provision for shutdown, unless it can be argued with absolute conviction that the system will never be blocked indefinitely in a waiting state at such points.

3.2.2. Backward-Directed Communication

At this point, we return to an examination of alternative approaches to interprocess communication within the framework of our example system. We shall now consider the second of the simple structures originally proposed in Fig. 3-2, depicted as follows:



This pattern of communication was referred to as "backward-directed," because the flow of control goes in the opposite direction from the flow of information. Thus, in constrast to the forward-directed pattern adopted initially, the downstream processes have the initiative here; in effect, they actively solicit successive data items by calling on their respective suppliers, rather than passively accepting delivery from upstream. On an intuitive basis, this particular approach would appear to be every bit as plausible as the one we first considered.

It can be seen that the configuration shown above is just a mirror image of the system which resulted from our previous decision to structure all communications in a forward-directed fashion. As such, the outside processes have merely switched roles: the Consumer has become an activity whereas the Producer in now a resource; as before, the Transducer is neither an activity nor a resource, but continues to be cast in its intermediary role.

From the communication pattern established by the above diagram, we can once more derive a correpsonding programmatic formulation wherein the constituent processes of the system are directly represented in terms of Ada tasks. These would now be specified (still within the declarative part of the main program Application) by the following declarations:

```
task Producer is
   entry Recv(A:out ARG);
end;

task Transducer is
   entry Recv(R:out RES);
end;

task Consumer;
```

3.2.2.1. Steady-State Operation

We shall again proceed by first developing a simple definition for the steady-state operation of our system. The following task bodies will serve to exhibit the backward-directed pattern of communication considered here:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      accept Recv(A:out ARG) dc
         A := AA;
      end;
   end loop;
end Producer;

task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      Producer.Recv(AA);
      Transform(AA,RR);
      accept Recv(R:out RES) do
         R := RR;
      end;
   end loop;
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
begin
   loop
      Transducer.Re  (RR);
      Dispose(RR);
   end loop;
end Consumer;
```

It can be seen that this steady-state definition is essentially identical to that which was first developed for the forward directed pattern. Only the entry call and accept statements are "inverted," reflecting the reversed direction of communication. As might be expected, the copies associated with transmitting data between processes still cannot be avoided except by sacrificing all logical concurrency.

3.2.2.2. Startup and Shutdown

Having thus developed a second formulation for the steady-state operation of our example system, we must once more address the problems of startup and shutdown for this alternative, backward-directed configuration. As with the forward-directed approach introduced initially, it turns out that startup requires no special attention. All three constituent processes of the system defined above are initiated implicitly, as a consequence of the corresponding task declarations. The order of initiation is again immaterial, since these tasks communicate solely by means of entry calls. This discipline ensures that the necessary synchronization will be established from the outset, whatever the direction of that communication.

With respect to shutdown, however, we shall encounter a rather pleasant surprise in the present context, at least when compared to the potential problems associated with the pattern of communication considered previously. Of course, there is still a need to adopt some basis for terminating execution of the Application program as a whole (since the steady-state definition for each of its component tasks involves an indefinite loop). For the time being, we shall continue to consider only a basic input-driven strategy, while maintaining our requirement that the overall system shut down gracefully (having completely processed all data items in the pipeline). The determination as to whether the input has been exhausted will be expressed here by the same simple expedient, a local loop counter:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   for N in range ... loop
      Acquire(AA);
      accept Recv(A:out ARG) do
         A := AA;
      end;
   end loop;
end Producer;
```

The surprise is that some such provision within the Producer process is sufficient -- the system will now shut down exactly as desired! Despite the fact that both the Transducer and Consumer tasks appear to loop indefinitely, they will nonetheless terminate appropriately, albeit exceptionally. What happens is that each of the processes represented by these latter tasks will effectively "commit suicide" at the proper moment, by attempting to communicate with a process that has already terminated (or is about to do so). Thus, with this approach, it is the failure to accept further communications which provides the synchronization required for shutdown.

The solution suggested above depends upon a judicious use of both the tasking and the exception mechanisms of Ada. The interaction between these separate language facilities can be summarized as follows:

- When a task terminates (regardless of how), all pending and future calls to any of its entries will result in the raising of a built-in exception (specifically, TASKING-ERROR) in the context of each such caller; in general, exceptions serve to notify a calling task of any operation that fails to complete successully.

- When this exception (like any other) is not explicitly handled at some level by the calling task, all of its outstanding operations are successively terminated; this signal is thus propagated back to the level of the task for which the exception was raised, but no further (i.e., not to any embedding tasks).

- When an unhandled exception (of any sort) reaches the level of the corresponding task body, execution of this latter is also terminated exceptionally; at the level of the immediately embedding task, however, there is no further distinction between normal and exceptional termination of its component subtasks.

These conventions of Ada can be exploited to great advantage in the present context, because raising of this exception then serves not only to signal the end of a given data stream but also to trigger termination of the receiving task when that event occurs. This is precisely the effect which is desired to achieve graceful shutdown in our example system as now formulated. The requisite properties may be established by the following argument:

1. The Producer process will deliver every input argument that it acquires to the Transducer, but when there are no more it will not accept any further communication and so terminate normally.

2. The Transducer process will deliver to the Consumer a result for every argument which it receives, but it will always request further data from the Producer and thus terminate exceptionally.

3. The Consumer process will dispose of every output result which it receives from the Transducer, but it will always request further data from that source and thus terminate exceptionally.

All of the constituent processes of our system will therefore complete action on the last item of information which enters the pipeline, and will eventually terminate (each in its own fashion). Termination of the corresponding component tasks (for whatever reason) means that execution of the main program Application will also terminate, whereupon the entire system will ultimately shut down just as we have required.

The relative simplicity of the solution obtained in this instance points up one of the primary advantages to what we have called a backward-directed pattern of interprocess communication (wherein data is effectively "pulled" rather than "pushed" downstream). When the receiving process is always actively requesting the next item of information, it is in a position to be directly informed of the success or failure of that request even if the transmitting process has already terminated. This is to be contrasted with the opposite approach, in which the receiver passively awaits the next item and so must somehow be informed (either explicitly or implicitly) that it should not wait any longer. The axiom of a system based on backward-directed communication is that the downstream processes are still alive so long as there are additional items to be transmitted (whence an isolated accept statement is not in any way dangerous); the transmitters are then structured to ensure that they will terminate on their side instead of waiting to accept further requests (so that their respective receivers will then be notified when they next attempt to solicit more input).

It should also be observed that the use of an exception to achieve this notification (and thereby trigger termination) is not at all asynchronous, since its raising is a potential consequence of the caller's own actions rather than of concurrent action by another process (as would be the case if the server, for instance, were to raise the FAILURE exception in the caller). Moreover, it is still possible for the caller to "trap" this exception at some level, and so perform certain finalization activities upon receiving such a signal (or even override its own termination):

```
task body Caller is
   ...
begin
   ...
   begin
      ...
      Server.Call(...);
      ...
   exception
      when others => ...
   end;
   ...
end Caller;
```

(No such possibility is present when the roles are reversed, and the built-in terminate signal is used). Finally, this "suicide" method of termination can certainly be regarded as distributed, in that it involves transactions between individual communicating processes, not some form of centralized knowledge. For these reasons, we consider this approach to be a legitimate design option, and see no need to introduce additional explicit signals in order to achieve shutdown in this context.

3.2.3. Inward-Directed Communication

We now turn to the third of the simple interprocess communication structures proposed in Fig. 3-2, depicted as follows:

The above pattern of communication was referred to as "inward-directed," because the flow of control goes from the outside in (whereas the flow of information is still from left to right). It is therefore the first of the "symmetric" communication patterns to be considered, as compared with those adopted previously, which were both asymmetric by nature. Within such a structure, the initiative is shared by the Producer and Consumer processes, while the Transducer process is essentially passive. This particular approach might be appropriate, for instance, in cases where the latter corresponds to some general-purpose utility operation -- e.g., a complex transformation that would more typically be predefined in an application library module for subsequent use in several different systems.

Some such hypothesis is in fact reflected in our characterization of the configuration shown above: here, both the Producer and the Consumer are to be regarded as activities, but the Transducer has become a pure resource; thus, there are no longer any processes which play an intermediary role in this overall organizational structure.

From the pattern specified by the above diagram we may again proceed directly to a corresponding Ada formulation. The constituent processes of our example system would now be represented by the following task declarations:

```
task Producer;

task Transducer is
   entry Send(A:in ARG);
   entry Recv(R:out RES);
end;

task Consumer;
```

3.2.3.1. Steady-State Operation

As before, we shall first establish the steady-state operation for our system, so as to exhibit the inward-directed communication pattern which is of interest here. This can be most simply obtained by defining the associated task bodies as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ...;
begin
   loop
      Acquire(AA);
      Transducer.Send(AA);
   end loop;
end Producer;


task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(AA:in ARG; RR:out RES) is ...;
begin
   loop
      accept Send(A:in ARG) do
         AA := A;
      end;
      Transform(AA,RR);
      accept Recv(R:out RES) do
         R := RR;
      end;
   end loop;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ...;
begin
   loop
      Transducer.Recv(RR);
      Dispose(RR);
   end loop;
end Consumer;
```

In this formulation, the Producer and Consumer tasks contain only entry calls (reflecting their status as pure activities), while both of the corresponding accept statements appear within the Transducer task (which is therefore a resource since it is the server for each of these calls).

This initial definition for the steady-state operation of our example has the same property which was present in previous formulations, namely, that the transmission of information between processes is accomplished by explicit copies (from transmitter to receiver), so as to maximally decouple the constituent processes and thereby achieve the greatest possible degree of logical concurrency within the system as a whole. However, in the context of symmetric patters of communication like that considered here, it may sometimes be appropriate to examine various design options in order to explore potential tradeoffs in this domain. By way of illustration, let us imagine two different definitions for the Transducer task, which might equally well have been introduced in place of that given above:

```
task body Transducer is  --Option 1
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ...;
begin
   loop
      accept Send(A:in ARG) do
         Transform(A,RR);
      end;
      accept Recv(R:out RES) do
         R := RR;
      end;
   end loop;
end Transducer;

task body Transducer is  --Option 2
   AA: ARG;
   procedure Transform(A:in ARG; R:out RES) is ...;
begin
   loop
      accept Send(A:in ARG) do
         AA := A;
      end;
      accept Recv(R:out RES) do
         Transform(AA,R);
      end;
   end loop;
end Transducer;
```

For Option 1, the Producer task is held in rendezvous with the Transducer while the argument data is transformed, thus avoiding the need to copy this information on input; however, the corresponding result is still transmitted by an explicit copy from the Transducer to the Consumer. Option 2 is just the opposite. Thus, in the first option, the combined operation of data acquisition and its ensuing transformation conceptually proceeds in parallel with that of the final data disposition, whereas the overlap is reversed in the second option.

3.2.3.2. Startup and Shutdown

We must once more complete our initial steady-state definition by addressing the issues of startup and shutdown for the inward-directed configuraion now under consideration. As for both the forward- and backward-directed approaches already discussed, startup poses no special problems. Thus we need only be concerned with how to achieve shutdown in the present context. Insofar as the current pattern of communication is a combination of the previous ones, we should expect to arrive at a solution which embodies certain aspects of each. Again, we shall confine our attention for the moment to a simple input-driven strategy, and continue to represent this determination by a local loop counter:

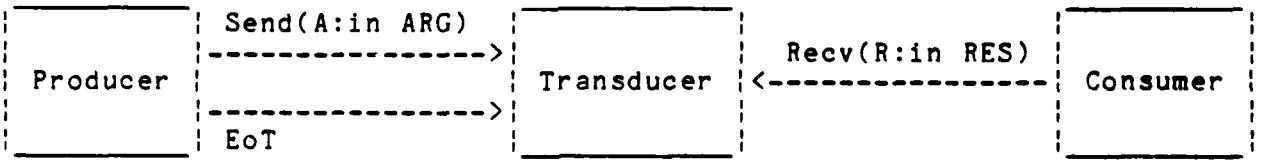
```
task body Producer is
   AA: ARG;
   procedure Acquire(A:in ARG) is ...;
begin
   for N in range ... loop
      Acquire(AA);
      Transducer.Send(AA);
   end loop;
end Producer;
```

Since we still want to shut our system down gracefully (i.e., process all items in the pipeline before proper termination), we must now focus on the downstream processes. With regard to the transactions between the Producer and the Transducer their pattern of communication is in effect forward-directed, and so it might seem that the "path of least resistance" would be to make use of the built-in termination mechanism:

```
task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ...;
begin
   loop
      select
         accept Send(A:in ARG) do
            AA := A;
         end;
         Transform(AA:RR);
         accept Recv(R:out RES) do
            R := RR;
         end;
      or
         terminate;
      end select;
   end loop;
end Transducer;
```

With regard to the pattern of communication between the Transducer and the Consumer, however, it can be seen that their transactions are essentially backward-directed. As such, it might be hoped that things would work out as for the approach where the entire system was based on a backward-directed pattern -- i.e., that the Consumer will terminate exceptionally by attempting to communicate with the Transducer once this latter has terminated. Unfortunately, we are in for a very rude shock: this will not work at all, because the Transducer task has not in fact terminated; rather, it is merely waiting to do so once all other tasks at this level have either terminated or are also waiting on such an alternative. But in this situation, the Consumer task is neither terminated nor in a quiescent state. Instead, it is actively soliciting further data from the Transducer (and thus waiting for that entry call to be completed). Hence, the system as a whole will eventually become suspended in the state described, and so never shut down. This inevitable deadlock provides a very vivid illustration of the pitfalls associated with the implicit termination mechanism of Ada. In particular, it serves to underscore the need to reason on a global basis (considering all component subtasks) whenever this would-be convenience is employed anywhere in the definition of a given system. Having learned a lesson, we shall henceforth forgo the use of this feature altogether.

We are nonetheless left with a quite viable option for achieving graceful shutdown, which is to introduce additional termination signals where necessary (for forward-directed transactions); since this approach allows the termination of a downstream process to be programmed explicitly, we may then rely on exceptional termination in the context of backward-directed transactions. Thus, our overall, inward-directed configuration would then be respecified as follows:



This diagram gives rise to the following task declarations and bodies:

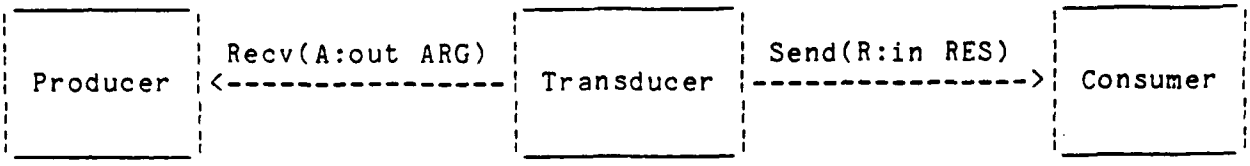
```
task Producer;

task Transducer is
   entry Send(A:in ARG);
   entry EoT;
   entry Recv(R:out RES);
end;

task Consumer;
```

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ...;
begin
   for N in range ... loop
      Acquire(AA);
      Transducer.Send(AA);
   end loop;
   Transducer.EoT;
end Producer;

task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ...;
begin
   loop
      select
         accept Send(A:in ARG) do
            AA := A;
         end;
         Transform(AA,RR);
         accept Recv(R:out RES) do
            R := RR;
         end;
      or
         accept EoT; exit;
      end select;
   end loop;
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ...;
begin
   loop
      Transducer.Recv(RR);
      Dispose(RR);
   end loop;
end Consumer;
```

This definition does indeed combine aspects of both the forward-directed solution (based on explicit signals) and the backward-directed solution (involving exceptional termination) in order to shut down the overall system, just as we expected at the outset. (Analogous modifications could easily be introduced in the context of optional formulations for the Transducer as discussed above, if it were desired to partially restrict the degree of logical concurrency within the system).

3.2.4. Outward-Directed Communication

We shall now consider the fourth and last of the simple structures for our example system which were originally proposed in Fig. 3-2, depicted as follows:



This pattern of communication was referred to as "outward-directed," because the flow of control goes from the inside out (though information still flows from left to right). Like the inward-directed approach just considered, the above pattern may also be said to be symmetric. It can be seen that in this instance, the initiative lies entirely with the Transducer process, since both the Producer and Consumer processes play an essentially passive role. Such an approach may seem, in some respects, to be the most intuitive of all, particularly if one were to think of the overall application program as being primarily embodied in the Transducer, and to view the Producer and Consumer solely as abstractions for some input source and output sink, respectively.

This viewpoint is once more reflected in our characterization of the configuration shown above: the Transducer is now the only activity in the system, whereas both the Producer and the Consumer are pure resources; thus, no process plays the mixed role of an intermediary in this organizational structure.

As usual, the above diagram serves as a specification for the desired communication pattern, from which we may derive declarations for the Ada tasks representing the constituent processes in the present configuration:

```
task Producer is
   entry Recv(A:out ARG);
end;


task Transducer;


task Consumer is
   entry Send(R:in RES);
end;
```

3.2.4.1. Steady-State Operation

According to our well established method, we shall first define the steady-state operation of the system, so as to exhibit the outward-directed pattern under consideration. In their simplest form, the associated task bodies would be as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ...;
begin
   loop
      Acquire(AA);
      accept Recv(A:out ARG) do
         A := AA;
      end;
   end loop;
end Producer;


task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES);
begin
   loop
      Producer.Recv(AA);
      Transform(AA,RR);
      Consumer.Send(RR);
   end loop;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ...;
begin
   loop
      accept Send(R:in RES) do
         RR := R;
      end;
      Dispose(RR);
   end loop;
end Consumer;
```

This formulation is to be compared with that introduced initially for the inward-directed pattern considered previously: as might be expected, they differ only that the entry calls and accept statements have again been inverted, thereby reversing the direction of communication.

Consistent with our overall objectives, this first definition is specifically intended to preserve the highest possible degree of logical concurrency within the system as a whole, whence information is transmitted between communicating processes by means of explicit copies. Again, however, one might wish to consider coupling the operation of these processes more tightly in certain circumstances, so as to be able to avoid such copying. This could be selectively accomplished by reformulating either the Producer or the Consumer task bodies as follows:

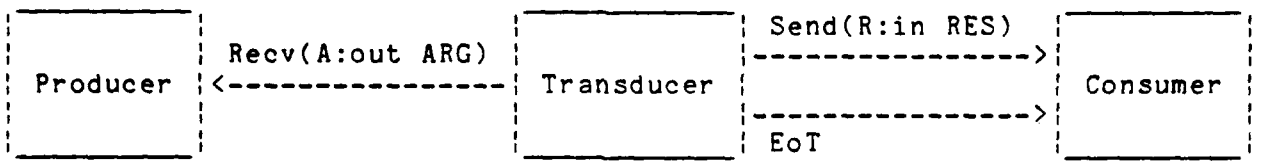
```
task body Producer is  --Option 1 .
   procedure Acquire(A:out ARG) is ...;
begin
   loop
      accept Recv(A:out ARG) do
         Acquire(A);
      end;
   end loop;
end Producer;


task body Consumer is  --Option 2
   procedure Dispose(R:in RES) is ...;
begin
   loop
      accept Send(R:in RES) do
         Dispose(R);
      end;
   end loop;
end Consumer;
```

With Option 1, the Transducer is held in rendezvous with the Producer while each new argument is acquired, thereby avoiding a copy of this information on input but sacrificing potential overlap of the data acquisition and transformation operations. Option 2, on the other side, is analogous: the Transducer is held in rendezvous with the Consumer while each successive result is disposed of, thereby avoiding a copy of this information on output but sacrificing potential overlap of the data transformation and disposition operations. From the standpoint of program design, it may or may not be advantageous to resort to one of these options in a given applications context. What must be emphasized, however, is that if <u>both</u> options are exercised together, then the operation of the overall system will in fact become completely sequential (and so might just as well have been formulated as a single process in the first place).

3.2.4.2. Startup and Shutdown

As always, we must now complete our definition for the system configuration currently considered by adapting the initial steady-state formulation so as to address the problems of startup and shutdown. Startup again requires no further attention, whence we need only be concerned with graceful shutdown.

Keeping still to the simple input-driven strategy adopted thusfar, we know that we must begin by first making suitable provisions within the Producer process, such that the corresponding task will terminate normally when all input arguments have been transmitted. Looking then to the downstream processes, we should like to be able to reason in terms of the separate transactions involved, between the Producer and the Transducer and between the Transducer and the Consumer, respectively. As for the former, we observe that their pattern of communication is backward-directed, whereupon we may make use of the exception mechanism to trigger termination of the Transducer task (once the Producer refuses to accept any further communication). With regard to the communication between the Transducer and the Consumer, their transactions are forward-directed. As such, we must either rely upon the built-in termination mechanism of Ada, or introduce an explicit signal in order to properly terminate the Consumer task. Having advised against use of the implicit mechanism (even though it would work in this particular instance), we shall thus opt here for explicitly programmed termination. The overall specification for the present system configuration may therefore be established as follows:



The corresponding task declarations (adding the EoT entry to the Consumer) would therefore become:

```
task Producer is
   entry Recv(A:out ARG);
end;

task Transducer;

task Consumer is
   entry Send(R:in RES);
   entry EoT;
end;
```

The constituent processes would then finally be defined as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ...;
begin
   for N in range ... loop
      Acquire(AA);
      accept Recv(A:out ARG) do
         A := AA;
      end;
   end loop;
end Producer;

task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ...;
begin
   loop
      Producer.Recv(AA); -- can fail!
      Transform(AA,RR);
      Consumer.Send(RR);
   end loop;
exception
   when others => Consumer.EoT;
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ...;
begin
   loop
      select
         accept Send(R:in RES) do
            RR := R;
         end;
         Dispose(RR);
      or
         accept EoT; exit;
      end select;
   end loop;
end Consumer;
```

By explicitly handling the exception raised by the Producer, the Transducer takes local responsibility for closing out its transactions with the Consumer; a further consequence of this approach is that all processes of the system turn out, in the end, to terminate normally.

3.3. Alternative Shutdown Strategies

In each of the simple patterns of interprocess communication considered up to this point, we have seen that our requirement for graceful shutdown of the overall system gave rise to a need for further synchronization. As such, the problem of proper termination always played a crucial role in establishing the final program design. In several instances, it ultimately led us to modify the initial pattern of communication by adding explicit signals so as to be able to achieve the desired effect; in others, we were content to rely upon an implicit signal, corresponding to the built-in exception which is raised by trying to communicate with a task that has already terminated. Whereas the "best" solution differed according to the particular communication pattern chosen as a point of departure, with every approach our design was not complete until we had adequately addressed the problem of synchronous shutdown.

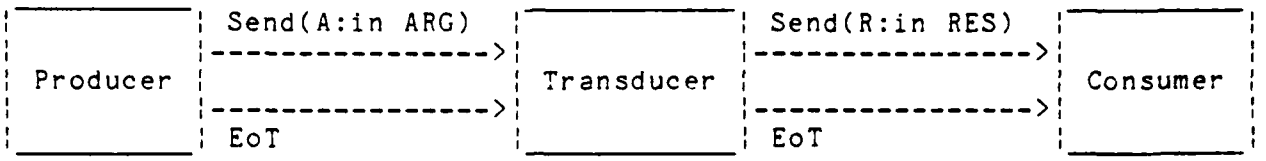
It should be remembered, however, that we have so far examined just one possible basis for shutting down the simple system taken as our example -- namely, to terminate execution of the entire application once it has somehow been determined that all available input items have been processed to completion (a strategy which we have thus referred to as "input-driven"). While this is certainly a reasonable basis, it must be recognized as only one amongst many alternative strategies that are commonly adopted in practice. Before attempting to draw any general conclusions in this regard, it would therefore seem advisable to explore various alternative approaches. In so doing, we shall abandon our usage of a simple loop counter, and instead express the termination decision in terms of a local predicate on the data itself, which determines whether some appropriate (but unspecified) "cuf-off" criterion has been attained. This (presumably more realistic) formulation would have appeared, in the context of the input-driven approaches already considered, as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ...;
   function CutOff(A:in ARG) return BOOLEAN is ...;
begin
   loop
      Acquire(AA);
      exit when CutOff(AA);
      ... [Transmit AA to Transducer] ...
   end loop;
   ...
end Producer;
```

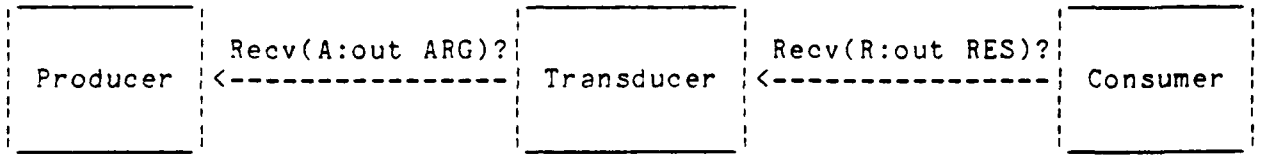
3.3.1. Input-Driven Strategy

In that this alternative was adopted as a basis for discussion of system shutdown in the previous section, a number of distinct solutions have already been developed there. These are summarized in Fig. 3-3.

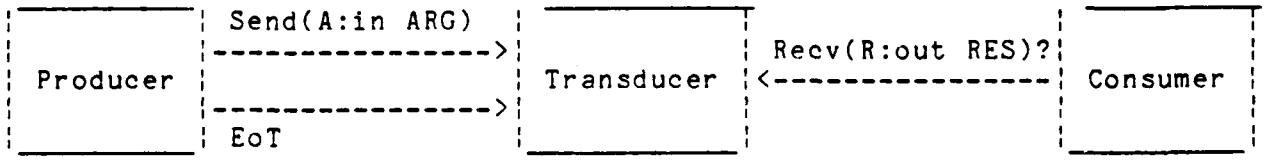
(1) Forward-Directed:



(2) Backward-Directed:



(3) Inward-Directed:



(4) Outward-Directed:

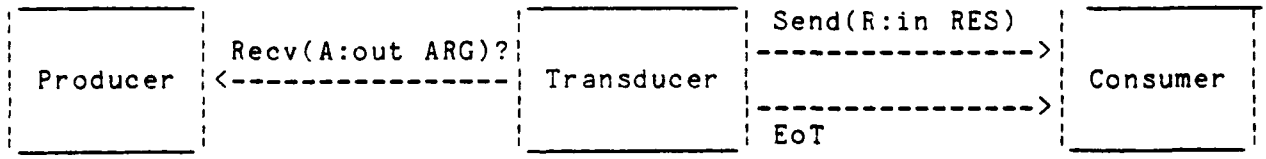


Figure 3-3: Possible Communication Patterns for Input-Driven Strategy.

The symbol "?" in the above specifications is a convention to be used from here on to indicate transactions that are expected to fail (i.e., to result in the raising of an exception) upon termination of the corresponding server process.

3.3.2. Output-Driven Strategy

We wish now to explore another equally common strategy for shutting down pipeline systems like the present example, namely to terminate execution once some cut-off criterion has been detected on the output side. We therefore refer to this alternative as "output-driven," in that the determination is made by the process which lies at end of the line. Thus, in this instance the decision is to be embodied in the Consumer, which will have the following skeleton form:

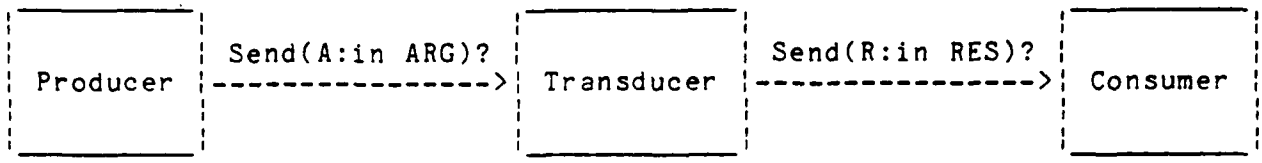
```
task body Consumer is
   RR: RES:
   procedure Dispose(R:in RES) is ... ;
   function CutOff(R:in RES) return BOOLEAN is ... ;
begin
   loop
      ... [Receive RR from Transducer] ...
      exit when CutOff(RR);
      Dispose(RR);
      ...
   end loop;
   ...
end Consumer;
```

The fundamental problem which arises in this context is that a suitable termination signal must somehow be propagated back upstream -- against the flow of information -- so as to systematically shut down the processes that are passing along, and ultimately generating, the data items in the pipeline.

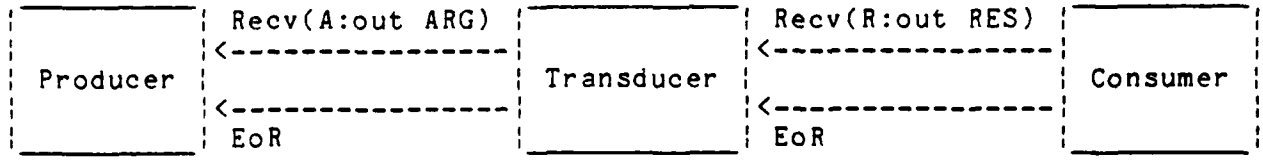
Whereas we still want to impose a requirement for graceful shutdown of our overall system, as we did in the context of a simple input-driven approach, some care must now be taken to define exactly what we mean by gracefully. Previously, we required that all items which entered the pipeline were processed to completion before proper termination. If we were to make the same demand for an output-driven strategy, it would have one of two implicatons: either the operation of these communicating processes must be so tightly coupled that the ultimate consumer can immediately cut off further data generation at the source (which would effectively preclude any logical concurrency whatsoever); or, alternatively, the final process must be prepared to accept additional data items, after the cut-off condition has been detected, so as to allow the pipeline to be progressively drained. We choose to reject both of these possibilities, and instead to require only that no further items will be received by the Consumer and that all of the constituent processes will eventually terminate in a proper fashion, such that the system as a whole can be shut down synchronously. Hence, in terms of the bucket-brigade analogy, we are willing to admit that "some water will be spilled" once the decision has been made.

From our previous experience in the context of an input-driven strategy, we know that there are in fact four different solutions which might be considered, based on the particular pattern of communication which is adopted for the steady-state operation of the system as a whole. These four possibilities are as specified in Fig. 3-4.

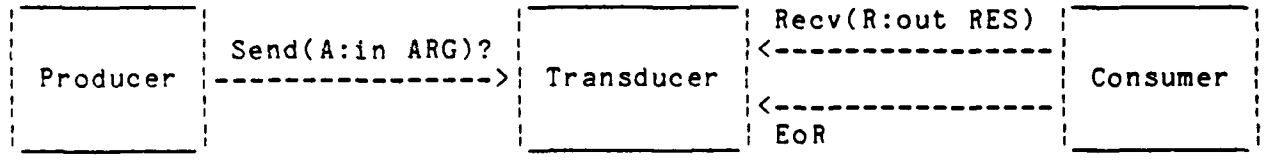
(1) Forward-Directed:



(2) Backward-Directed:



(3) Inward-Directed:



(4) Outward-Directed:

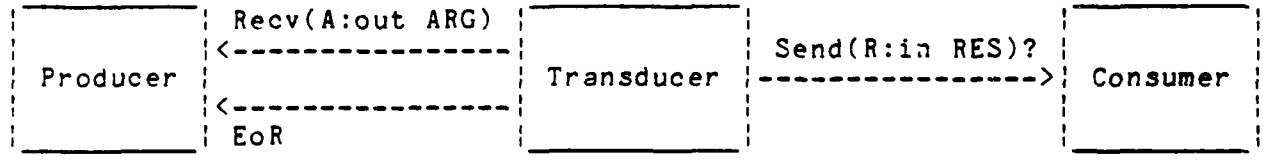


Figure 3-4: Possible Communication Patterns for Output-Driven Strategy.

It may be observed that each configuration depicted in Fig. 3-4 corresponds to the mirror image of that for the opposite communication pattern in the input-driven context (cf. Fig. 3-3), which is indicative of the true relationship between these alternative shutdown strategies.

From the specifications in Fig. 3-4, we can directly derive the corresponding task declarations for each of these four solutions:

(1) Forward-Directed:

```
task Producer;      -- an activity

task Transducer is
   entry Send(A:in ARG);
end;

task Consumer is    --  a resource
   entry Send(R:in RES);
end;
```

---

(2) Backward-Directed:

```
task Producer is    --  a resource
   entry Recv(A:out ARG);
   entry EoR;
end;

task Transducer is
   entry Recv(R:out RES);
   entry EoR;
end;

task Consumer;      -- an activity
```

---

(3) Inward-Directed:

```
task Producer;      -- an activity

task Transducer is --  a resource
   entry Send(A:in ARG);
   entry Recv(R:out RES);
   entry EoR;
end;

task Consumer;      -- an activity
```

---

(4) Outward-Directed:

```
task Producer is    --  a resource
   entry Recv(A:out ARG);
   entry EoR;
end;

task Transducer;    -- an activity

task Consumer is    --  a resource
   entry Send(R:in RES);
end;
```

We shall now present the associated process definitions for all four solutions in turn, formulated so as to maximize the logical concurrency.

(1) For the forward-directed pattern, the definitions are as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      Transducer.Send(AA); -- ?
   end loop;
end Producer;

task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      accept Send(A:in ARG) do
         AA := A;
      end;
      Transform(AA,RR);
      Consumer.Send(RR);  -- ?
   end loop;
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
   function CutOff(R:in RES) return BOOLEAN is ... ;
begin
   loop
      accept Send(R:in RES) do
         RR := R;
      end;
      exit when CutOff(RR);
      Dispose(RR);
   end loop;
end Consumer;
```

The above definitions directly exhibit how the shutdown signal is propagated back upstream (and the "spillage" involved). Once a result is received which satisfies the cut-off criterion, it is not disposed of by the Consumer, but rather this process simply terminates. In consequence, the next (already transformed) item in the pipeline will not be successfully transmitted by the Transducer, but instead this process will also terminate (exceptionally). Finally, the same fate awaits the last argument acquired by the Producer, where an unsuccessful attempt to transmit that item again leads to exceptional termination -- and thus to shutdown of the entire system.

(2) For the backward-directed pattern, the definitions are as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      select
         accept Recv(A:out ARG) do
            A := AA;
         end;
      or
         accept EoR; exit;
      end select;
   end loop;
end Producer;

task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      Producer.Recv(AA);
      Transform(AA,RR);
      select
         accept Recv(R:out RES) do
            R := RR;
         end;
      or
         accept EoR; exit;
      end select;
   end loop;
   Producer.EoR;
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
   function CutOff(R:in RES) return BOOLEAN is ... ;
begin
   loop
      Transducer.Recv(RR);
      exit when CutOff(RR);
      Dispose(RR);
   end loop;
   Transducer.EoR;
end;
```

(3) For the inward-directed pattern, the definitions are as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      Transducer.Send(AA); -- ?
   end loop;
end Producer;

task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      accept Send(A:in ARG) do
         AA := A;
      end;
      Transform(AA,RR);
      select
         accept Recv(R;out RES) do
            R := RR;
         end;
      or
         accept EoR; exit;
      end select;
   end loop;
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
   function CutOff(R:in RES) return BOOLEAN is ... ;
begin
   loop
      Transducer.Recv(RR);
      exit when CutOff(RR);
      Dispose(RR);
   end loop;
   Transducer.EoR;
end;
```

The difference between the input-driven and output-driven strategies can perhaps best be seen here by comparing the structure of the Transducer as defined above with its corresponding formulation at the end of Section 3.2.3.2 (note, however, that their steady-state operation is in fact identical).

(4) For the outward-directed pattern, the definitions are as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      select
         accept Recv(A:out ARG) do
            A := AA;
         end;
      or
         accept EoR; exit;
      end select;
   end loop;
end Producer;

task body Transducer is
   AA: ARG;
   RR: RES;
   procedure Transform(A:in ARG; R:out RES) is ... ;
begin
   loop
      Producer.Recv(AA);
      Transform(AA,RR);
      Consumer.Send(RR);  -- ?
   end loop;
exception
   when others => Producer.EoR
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
   function CutOff(R:in RES) return BOOLEAN is ... ;
begin
   loop
      accept Send(R:in RES) do
         RR := R;
      end;
      exit when CutOff(RR);
      Dispose(RR);
   end loop;
end Consumer;
```

The difference between the input- and output-driven shutdown strategies is even more evident here when the Transducer as defined above is compared with its counterpart as formulated at the end of Section 3.2.4.2.

3.3.3. Transit-Driven Strategy

Another shutdown strategy that is typically encountered in the context of pipeline systems like our present example (and the last to be considered in detail here) is one in which the termination decision is made somewhere in passage, by one of the processes serving to convey successive data items from their original source to their final destination. We therefore refer to this particular alternative as "transit-driven."

Within the current framework, such a strategy necessarily implies that the determination must be made by the Transducer process. For purposes of illustration, we shall adopt yet another approach for programming the actual decision -- namely to add an additional output parameter to the transformation procedure, which serves as a "status return" (i.e., indicating whether or not the transform in question was successful, according to some purely internal criterion). Thus, the definition of the Transducer process will then appear in skeleton form as follows:

```
task body Transducer is
   AA: ARG;
   RR: RES;
   OK: BOOLEAN;
   procedure Transform(A:in ARG; R:out RES; S:out BOOLEAN) is ...;
begin
   loop
      ... [Receive AA from Producer] ...
      Transform(AA, RR, OK);
      exit when not OK;
      ... [Transmit RR to Consumer] ...
   end loop;
   ...
end Transducer;
```

Expressed in this fashion, the Transducer may be viewed as a "valve," in that it acts to shut off the flow of information once a certain (locally defined) termination condition has been detected.

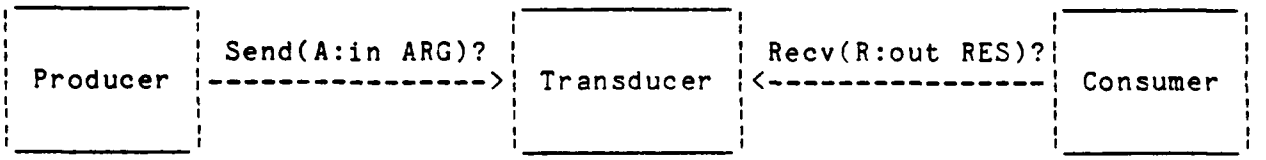
The strategy now considered can be seen as more of a mixed one, combining aspects of both the input-driven and output-driven approaches introduced previously. In particular, the application should be expected to behave like an output-driven system on the upstream side (before the decision is made) and like an input-driven system downstream from there. Hence the requirement for graceful shutdown to be imposed here will be taken to exactly reflect these expectations.

Again, we know that there are in fact four separate solutions, all of which might be deemed equally acceptable. These possibilities, based on different patterns of communication for the steady-state operation of the overall system, are as specified in Fig. 3-5.

(1) Forward-Directed:

(2) Backward-Directed:

(3) Inward-Directed:



(4) Outward-Directed:

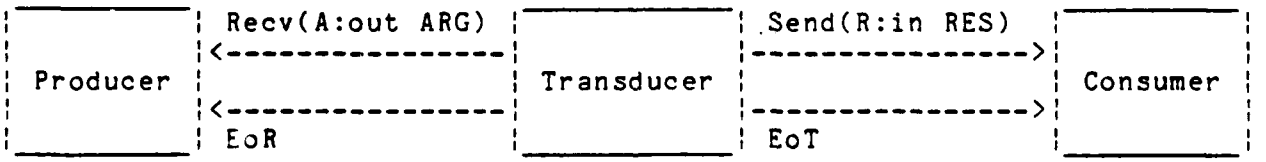


Figure 3-5: Possible Communication Patterns for Transit-Driven Strategy.

The configurations specified above do indeed depict the mixed nature of a transit-driven shutdown strategy: in all four patterns, the transactions on the left are the same as for the output-driven approach (cf. Fig. 3-4), whereas those on the right are the same as for the input-driven case (cf. Fig. 3-3).

We can again directly derive the corresponding task declarations for each of these four solutions from their specifications in Fig. 3-5.

(1) Forward-Directed:

```
task Producer;      -- an activity

task Transducer is
   entry Send(A;in ARG);
end;

task Consumer is    -- a resource
   entry Send(R:in RES);
   entry EoT;
end;
```

---

(2) Backward-Directed:

```
task Producer is    -- a resource
   entry Recv(A:out ARG);
   entry EoR;
end;

task Transducer is
   entry Recv(R:out RES);
end;

task Consumer;      -- an activity
```

---

(3) Inward-Directed:

```
task Producer;      -- an activity

task Transducer is -- a resource
   entry Send(A:in ARG);
   entry Recv(R:out RES);
end;

task Consumer;      -- an activity
```

---

(4) Outward-Directed:

```
task Producer is    -- a resource
   entry Recv(A:out ARG);
   entry EoR
end;

task Transducer;    -- an activity

task Consumer is    -- a resource
   entry Send(R:in RES);
   entry EoT;
end;
```

We once more present the associated process definitions for all four solutions, formulated as before so as to maximize logical concurrency.

(1) For the forward-directed pattern, the definitions are as follows:

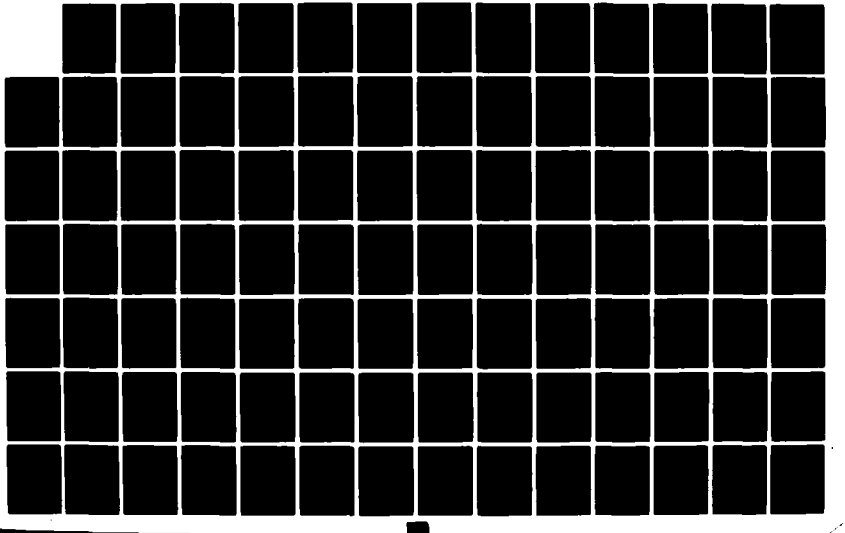
```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire (AA);
      Transducer.Send(AA); -- ?
   end loop;
end Producer;


task body Transducer is
   AA: ARG;
   RR: RES;
   OK: BOOLEAN;
   procedure Transform(A:in ARG; R:out RES; S:out BOOLEAN) is ... ;
begin
   loop
      accept Send(A:in ARG) do
         AA := A;
      end;
      Transform(AA, RR, OK);
      exit when not OK;
      Consumer.Send(RR);
   end loop;
   Consumer.EoT;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in R  is ... ;
begin
   loop
      select
         accept Send(R:in RES) do
            RR := R;
         end;
         Dispose (RR);
      or
         Accept EoT; exit;
      end select;
   end loop;
end Consumer;
```

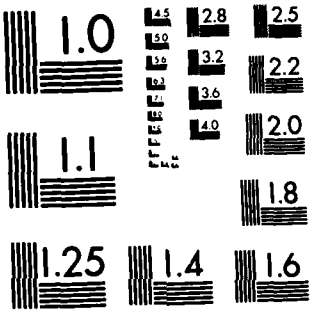
(2) For the backward-directed pattern, the definitions are as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      select
         accept Recv(A:out ARG) do
            A := AA;
         end;
      or
         accept EoR; exit;
      end select;
   end loop;
end Consumer;


task body Transducer is
   AA: ARG;
   RR: RES;
   OK: BOOLEAN;
   procedure Transform(A:in ARG; R:out RES; S:out BOOLEAN) is ... ;
begin
   loop
      Producer.Recv(AA);
      Transform(AA, RR, OK);
      exit when not OK;
      accept Recv(R:out RES) do
         R := RR;
      end;
   end loop;
   Producer.EoR;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
begin
   loop
      Transducer.Recv(RR); -- ?
      Dispose(RR);
   end loop;
end Consumer;
```

AD-A124 012 USING SELECTED FEATURES OF ADA: A COLLECTION OF PAPERS (U) BATTELLE COLUMBUS LABS OH N HABERMAN ET AL. 09 NOV 82 DAAG29-76-D-0100 2/3

UNCLASSIFIED F/G 9/2 NL





(3) For the inward-directed pattern, the definitions are as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire (AA);
      Transducer.Send(AA); -- ?
   end loop;
end Producer;


task body Transducer is
   AA: ARG;
   RR: RES;
   OK: BOOLEAN;
   procedure Transform(A:in ARG; R:out RES; S:out BOOLEAN) is ... ;
begin
   loop
      accept Send(A:in ARG) do
         AA := A;
      end;
      Transform(AA, RR, OK);
      exit when not OK;
      accept Recv(R:out RES) do
         R := RR;
      end;
   end loop;
end Transducer;


task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
begin
   loop
      Transducer.Recv(RR); -- ?
      Dispose(RR);
   end loop;
end Consumer;
```

(4) For the outward-directed pattern, the definitions are as follows:

```
task body Producer is
   AA: ARG;
   procedure Acquire(A:out ARG) is ... ;
begin
   loop
      Acquire(AA);
      select
         accept Recv(A:out ARG) do
            A := AA;
         end;
      or
         accept EoR; exit;
      end select;
   end loop;
end Consumer;

task body Transducer is
   AA: ARG;
   RR: RES;
   OK: BOOLEAN;
   procedure Transform(A:in ARG; R:out RES; S:out BOOLEAN) is ... ;
begin
   loop
      Producer.Recv(AA);
      Transform(AA, RR, OK);
      exit when not OK;
      Consumer.Send(RR);
   end loop;
   Producer.EoT;
   Consumer.EoR;
end Transducer;

task body Consumer is
   RR: RES;
   procedure Dispose(R:in RES) is ... ;
begin
   loop
      select
         accept Send(R:in RES) do
            RR := R;
         end;
         Dispose (RR);
      or
         Accept EoT; exit;
      end select;
   end loop;
end Consumer;
```

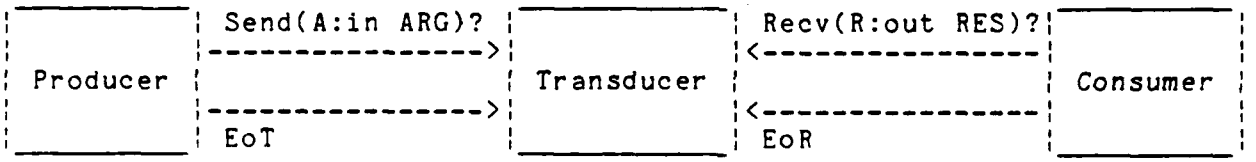
3.3.4. Other Possible Strategies

We have not yet exhausted all possible shutdown strategies. For instance, one could adopt an "extrema-driven" approach, wherein the termination decision is made by either the input or output process (applying completely independent criteria). This would give a "superposition" of those previous strategies, as specified in Fig. 3-6.

(1) Forward-Directed:

(2) Backward-Directed:

(3) Inward-Directed:



(4) Outward-Directed:

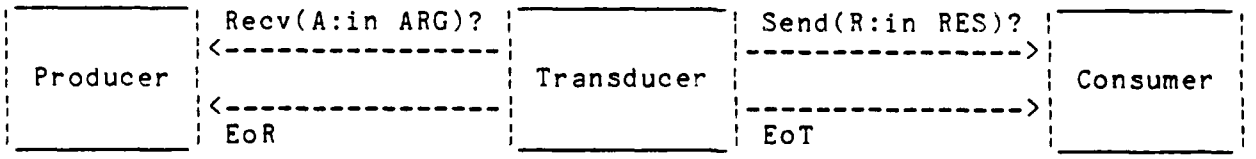


Figure 3-6: Possible Communication Patterns for Extrema-Driven Strategy.

We feel confident, however, that the programming of these (or any other) composite solutions in Ada can by now be left as an exercise for the reader.

TUTORIAL ON ADA EXCEPTIONS

by

David B. Loveman

30 March 1981

TUTORIAL ON ADA EXCEPTIONS

by

David B. Loveman

This tutorial describes Ada's facilities for dealing with exceptional situations, such as errors, and provides examples of the use of these facilities. "An exception is an event that causes suspension of normal program operation. Drawing attention to the event is called raising the exception. Executing some actions, in response to the occurrence of an exception, is called handling the exception." [LRM] Chapter 11.

THE ROLE OF EXCEPTIONS

The ability of a program to handle certain exceptional situations is essential. Such situations, typically but not necessarily errors, occur rarely, but are likely to happen given enough time. A survey, taxonomy, and presentation of language features for exceptions is given in [Goodenough]. Ada's approach is defined in the [LRM], Chapter 11, and discussed in the [Rationale], Chapter 12.

The Concept of Exceptions

[Goodenough] observes that, in general, an exception's full significance is known only outside the detecting operation; the operation cannot determine unilaterally what is to be done after an exception is raised.

> In essence, exceptions permit the user of an operation to extend an operation's domain (the set of inputs for which effects are defined) or its range (the effects obtained when certain inputs are processed). Exceptions permit a user to tailor an operation's results or effects to his particular purpose in using the operation. In short, exceptions serve to generalize operations, making them usable in a wider variety of contexts than would otherwise be the case. Specifically, exceptions are used:
>
> (a) to permit dealing with an operation's impending or actual failure. Two types of failure are of interest: range failure, and domain failure;
>
> (b) to indicate the significance of a valid result or the circumstances under which it was obtained.
>
> (c) to permit an invoker to monitor an operation, e.g. to measure computational progress or to provide

additional information and guidance should certain conditions arise.

[Goodenough] goes on to classify exceptions into three categories:

(a) ESCAPE exceptions, which require termination of the operation raising the exception;

(b) NOTIFY exceptions, which forbid termination of the operation raising the exception and require its resumption after the handler has completed its actions; and

(c) SIGNAL exceptions, which permit the operation raising the exception to be either terminated or resumed at the handler's discretion.

[Steelman] specifically requires that the occurrence of an exception cause a transfer to an appropriate handler without completion of the operation in which the exception occurred. Thus Ada exceptions are of the ESCAPE category only, and not of the NOTIFY or SIGNAL categories; they serve only for error situations and as terminating conditions. It is worth mentioning that the NOTIFY and SIGNAL capabilities for monitoring an operation can be realized in a straightforward manner utilizing rendezvous. The general topic of tasking in Ada is discussed by [Schuman].

The indication of the significance of a result can be implemented using Ada exceptions. Such result classification information is usually passed directly from callee to caller. As a result, a more natural implementation utilizes status variables as added parameters of call. An example is given in a later section.

Error Exceptions

Errors can be subdivided into domain errors and range errors. A domain error occurs when the inputs to an operation fail to pass some input assertion as to their acceptability. An example of such an input assertion is the requirement that, in an operation to add together two matrices, both matrices must be of the same size. A typical response to a domain error is an attempt to "correct" the inputs and try the operation again. Typically a domain error is detected before the operation performs any actions which need to be undone.

A range error occurs when an operation determines that its output assertion for determining the validity of its result may not be satisfied. This may occur in two ways, either definite failure of the output assertion, or evidence that it can never be satisfied. [Goodenough] gives examples of these errors:

definite failure
   end of file on read
   divergence in a numerical algorithm

evidence of failure
   parity error on read
   lack of convergence after a fixed amount of effort.

An operation in which a range error occurs will, in general, have proceeded to the point where some side effects of the operation will need to be undone.

Both errors require the ability to terminate an operation prematurely, perhaps allowing the operation the right to "clean up" after itself. In the next section we shall review Ada's language features for exception handling. Following that we shall discuss an approach to the systematic use of exceptions, and present several examples.

ADA EXCEPTION HANDLING

The name of an exception is declared by means of an exception declaration. An exception occurs as a result of being explicitly raised by a raise statement or, more typically, as a result of being propagated by subprograms, blocks, or language defined operations. Exceptions are handled by user-written exception handlers placed at the end of a block or subprogram, package or task body. When an exception is raised, the execution of a handler replaces the execution of the unit in which the exception occurred. The choice of a particular handler for an exception is dynamic and can, in general, only be determined at run time.

Basic Features

A simple example showing the use of some of Ada's exception facilities is given below:

```
...
MY_ERROR: exception;
...
begin
   ...
   if SOME_CONDITION then
      raise MY_ERROR;
   endif;
   ...
   if SOME_OTHER_CONDITION then
      raise MY_ERROR;
   endif;
   ...
exception
   when MY_ERROR =>
      DO_SOMETHING;
      raise YOUR_ERROR;
   when others =>
      DO_SOMETHING_ELSE;
      raise;
end;
```

The above example illustrates the declaring, raising, re-raising, and handling of an exception within a single piece of code, and the use of others to indicate the handling of any exception save the ones explicitly listed in the handler. The example does not illustrate:

- built-in exceptions,
- suppression of exceptions, or
- propagation of exceptions.

We shall discuss these briefly now.

Built-in Exceptions

Ada predefines exceptions in two places, the package STANDARD and the packages INPUT_OUTPUT and TEXT_IO. Figure 1 summarizes briefly the exceptions from package STANDARD. These exceptions in general correspond to errors for which a simple fix is not possible. We shall see later however that there are cases for which a fix-up and re-try is possible. We are here not concerned with the use of the exception attribute FAILURE; its use is discussed in [Schuman].

Figure 2 summarizes the input/output exceptions. As with the STANDARD exceptions, these should be considered as errors, and not used in routine programming. For example, in order to copy one text file to another, one should, for a termination test, use the function END_OF_FILE and not the exception END_ERROR, as shown in Figure 3. We have ignored in this example the possibility that, as a result of a mistake or system failure, a USE_ERROR, STATUS_ERROR, or DEVICE_ERROR might occur; we shall discuss this more fully later.

CONSTRAINT_ERROR	Raised upon violation of range, index, or discriminant constraints; attempted reference to a non-existent record component; attempted reference through an access value of null.
NUMERIC_ERROR	Raised, for some machines, upon numeric overflow or underflow.
SELECT_ERROR	Raised when no alternative of a select statement is open.
STORAGE_ERROR	Raised when insufficient storage space remains for a task or new allocation.
TASKING_ERROR	Raised when exceptions occur during rendezvous.
'FAILURE	A task attribute which is an exception. Raising T'FAILURE causes an exception in task T.

Figure 1: Exceptions predefined in the language in package STANDARD

NAME_ERROR	Incorrect use of external file name File already exists on call to CREATE No such file exists on call to OPEN or DELETE
USE_ERROR	Operation incompatible with external file properties Attempt to lengthen a file via TRUNCATE Attempt to WRITE to a protected file
STATUS_ERROR	File not in proper status for an operation File already open on call to CREATE or OPEN File not open on call to CLOSE or READ No external file associated with internal file on call to NAME No default file on call to CURRENT_INPUT
DATA_ERROR	Value not defined on input Value not defined on call to READ Identifier not TRUE or FALSE on GET of BOOLEAN Identifier not one of the enumeration literals on GET of an enumeration type (N.b. On a GET of a numeric type, if the value is out of range, CONSTRAINT_ERROR is raised)
DEVICE_ERROR	Malfunction of underlying system
END_ERROR	Current read position is higher than end position Attempt to READ or GET past end of file
LAYOUT_ERROR	Incorrect text formatting Attempt to SET_COL greater than line length Attempt to PUT a string larger than the line

Figure 2: Exceptions defined in packages INPUT_OUTPUT and TEXT_IO.

Suppression of Exceptions

By use of the pragma SUPPRESS, the check for some of the conditions under which certain predefined exceptions are raised may be suppressed. A summary of these check conditions is presented in Figure 4.

Use of the pragma SUPPRESS is a recommendation to the compiler to avoid compiling run time checks. The programmer has assumed the burden of guaranteeing exception-freeness of the code since, should an exception occur whose run time check is suppressed, the results of the program will be unpredictable.

```
-- the wrong way to do it
procedure COPY(F:IN_FILE; G:OUT_FILE) is
   C: CHARACTER;
begin
   loop
      GET(F,C);
      PUT(G,C);
   end loop;
exception
   when END_ERROR => return;
end COPY;
```

As opposed to:

```
-- the right way to do it
procedure COPY(F:IN_FILE; G:OUT_FILE) is
   C: CHARACTER;
begin
   while not END_OF_FILE(F) loop
      GET(F,C);
      PUT(G,C);
   end loop;
   return;
end COPY;
```

Figure 3: END_OF_FILE in lieu of END_ERROR

SUPPRESSible Check	Programmer Guarantees
CONSTRAINT_ERROR exception	
ACCESS_CHECK	access value not null
DISCRIMINANT_CHECK	variant record component exists
INDEX_CHECK	index constraint satisfied
LENGTH_CHECK	proper number of components
RANGE_CHECK	range constraint satisfied
NUMERIC_ERROR exception	
DIVISION_CHECK	divisor not zero
OVERFLOW_CHECK	numeric operation does not overflow
STORAGE_ERROR exception	
STORAGE_CHECK	sufficient space is available

Figure 4: Checks which may be suppressed

Propagation of Exceptions

An exception may be raised in a program unit in two ways. It may be explicitly raised by a raise statement, or it may be propagated by program units, including operators, executed by the given program unit. For example, NUMERIC_ERROR will be raised both by the statement

```
raise NUMERIC_ERROR;
```

and by the expression

```
... 3/0 ...
```

Within a handler for the exception NUMERIC_ERROR, the exception may be reraised, and passed to a higher level program by the statement

```
raise NUMERIC_ERROR;
```

or the shorter

```
raise;
```

This second form is most convenient when processing anonymous exceptions in a handler for others.

Once an exception is raised, an appropriate handler for it is found according to the rules given in Chapter 11 of the LRM. Roughly speaking, the rules are as follows:

1. An exception raised in a declarative part during elaboration is propagated to the unit which caused the elaboration.

2. An exception raised in a sequence of statements is handled by a local handler if present in the innermost block or body enclosing the statement which raised the excepti n.

3. An exception for which there is no local handler, or which itself occurs within a handler, is propagated to the unit which caused execution of the current unit, except from a task to its invoker.

Figure 5 provides two examples which illustrate the difference between exceptions raised in a declarative part and exceptions raised in an executable part. In short, exceptions raised in a declarative part cannot be handled locally and must be handled by the invoker. Thus, in general, one must be careful in order to fully encapsulate abstract data types.

```
declare
   MY_ERROR: exception;
   ...
begin
   ...
   declare
      N: INTEGER := F();
   begin
      ...
   exception
      when MY_ERROR => ...    -- handler E1
   end;
   ...
exception
   when MY_ERROR => ...       -- handler E2
end;
-- if F raises MY_ERROR, it is handled by E2


declare
   MY_ERROR: exception;
   ...
begin
   ...
   declare
      N: INTEGER;
   begin
      N := F();
      ...
   exception
      when MY_ERROR => ...     -- handler E1
   end;
   ...
exception
   when MY_ERROR => ...        -- handler E2
end;
-- if F raises MY_ERROR, it is handled by E1
```

Figure 5: Exceptions in declarative and executable parts

USES OF EXCEPTIONS

This section presents a number of examples of the stylized use of exceptions, derived from [Rationale].

Multi-level Return

A typical problem involves a main program which controls the repetitive operation of some procedure. This procedure, in turn, can depend on subordinate procedures, any one of which might develop an error condition from which recovery is not possible. At the top level, however, recovery is possible by the expedient of abandoning the current data case and moving on to the next. In other words, whenever an exception occurs, execution should be abandoned and control transferred to a particular point. This is sometimes referred to as an "up-stack goto".

In the following example, procedure P treats 20 matrices. Treatment of a matrix involves reading it in, inverting it, and printing it out. The second level procedure INVERT may have a NUMERIC_ERROR exception, which it transforms to the exception SINGULAR. This is interpreted by procedure P as "abandon this case and go on to the next".

```
procedure P is
   SINGULAR: exception;
   procedure TREAT_A_MATRIX is
      procedure INVERT(M: out MATRIX) is
      begin
         -- may raise NUMERIC_ERROR
      exception
         when NUMERIC_ERROR =>
            raise SINGULAR;
      end INVERT;
   begin -- TREAT_A_MATRIX
      READ(M);
      INVERT(M);
      PRINT(M);
   end TREAT_A_MATRIX;
begin -- P
   for I in 1..20 loop
      PRINT("ITERATION ",I);
      begin
         TREAT_A_MATRIX;
      exception
         when SINGULAR =>
            PRINT(" singular -- on to the next case");
      end;
   end loop;
end P;
```

Clean Up

In a sequence of procedure calls, the occurrence of an exception causes termination of procedures in the dynamic call chain up to the first procedure handling the exception. Procedures along the way, although not fully processing the exception, may wish to express "last wishes" in order to perform clean-up actions. The handler for others, and the raise statement used to reraise the same exception in the calling environment, can be used to obtain this effect.

Consider, for example, a procedure which performs some file operation:

```
procedure FILE_OPERATION(FILE_NAME:STRING) is
   F: INOUT_FILE;
begin
   -- initial actions
   OPEN(F,FILE_NAME);
   -- perform work on the file
   CLOSE(F);
   -- final actions
end;
```

If an exception of any type should occur while the file is open, the procedure FILE_OPERATION will terminate, without closing the file.

This problem can be eliminated by rewriting the procedure, enclosing the work to be done, which might cause an exception, within a block. The block handles any exception by closing the file and reraising the exception.

```
procedure FILE_OPERATION(FILE_NAME:STRING) is
   F: INOUT_FILE;
begin
   -- initial actions
   OPEN(F,FILE_NAME);
   begin
      -- perform work on the file
   exception
      when others =>
         CLOSE(F);
         raise;
   end;
   CLOSE(F);
   -- final actions
end;
```


/c 8

Retry of an Operation

As pointed out earlier, certain types of range errors are, potentially, intermittent. For example, an attempt to read a tape block may result in a momentary error. On rereading, the operation may be a success. Given a low level READ_TAPE operation which potentially raises a TAPE_ERROR exception, we can write a loop which will retry the operation up to 10 times. If still unsuccessful, a higher level exception MALFUNCTION is raised. Note the use of a block to, in essence, scope exception handling to a single statement.

```
for I in 1..10 loop
   begin
      READ_TAPE(BLOCK);
      exit;
   exception
      when TAPE_ERROR =>
         if I = 10 then
            raise MALFUNCTION;
         else
            BACKSPACE;
         end if;
   end;
end loop;
```

Domain Extension

Exceptions which indicate domain errors can be used to, in effect, extend the domain of a preexisting operation. For example, "/" on a user defined floating point type REAL will not be defined for a denominator of zero. By means of overloading, the definition of "/" for REAL arguments can be extended:

```
function "/"(X,Y: REAL) return REAL is
begin
   return STANDARD."/"(X,Y);
exception
   when NUMERIC_ERROR =>
      return REAL'LARGE;
end;
```

EXCEPTIONS AND ABSTRACTIONS

Issues

A number of problems arise when one attempts to systematically use exceptions with packages in the implementation of abstractions. Ada exceptions allow considerable freedom, and may be used in ways which do not enhance the reliability of the code being produced. Three facts in particular about exceptions must be dealt with in order to structure their use:

1. Although exceptions are meant for error and other seldom occurring conditions, they can be utilized as a "normal" control structure. For example, one can program a block exit by means of exceptions:

```
declare
   BLOCK_EXIT: exception;
begin
   ...
   if SOME_CONDITION then raise BLOCK_EXIT; end if;
   ...
exception
   when BLOCK_EXIT => null;
end;
```

2. An exception, if it can occur, is clearly one of the potential external effects of a procedure or a package. There is no language-mandated requirement that it be documented as such. Clearly good programming style requires that any procedure or package which implements an abstraction must have all of its possible external effects documented. In the next section we shall provide guidance for such documentation which, if followed, will alleviate this and other problems.

3. An exception may be propagated beyond the scope of its name, and there it can be handled only by means of others. Indeed, an exception may be propagated beyond the scope of its name, and then back within its scope again. This is a consequence of the fact that the search for a handler follows the dynamic call chain and is illustrated by the following example from [Rationale]:

```
package D is
   procedure A;
   procedure B;
end;

procedure OUTSIDE is
begin
  ... D.A; ...
end;

package body D is
   ERROR: exception;
   procedure A is
   begin
      ... raise ERROR; ...
   end;
   procedure B is
   begin
   ... OUTSIDE; ...
   exception
      when ERROR =>
         -- ERROR may be propagated by OUTSIDE calling A
   end;
end D;
```

The Approach

The approach we recommend is fairly simple, and consists of three parts:

1. Code in such a manner that an exception never escapes an encapsulation without handling and recasting into a form appropriate for the abstraction. Thus an encapsulation boundary serves as a firewall for runaway exceptions and automatic propogation.

2. To assist with 1, code in a style which always specifically names user exceptions, rather than depending on others. As a firewall, at least all enapsulation boundaries will have handlers for others. One style is to have a single, program_wide exception GLOBAL_ERROR, which can be handled explicitly when appropriate. The encapsulation firewall will consist of

```
...
exception
   ...
   when others => raise GLOBAL_ERROR;
end;
```


///

3. Carefully document the use of all exceptions. Our recommendation for this is presented in the next section.

Exception Documentation

[Luckham] has observed that if selected assertions and exception propagation declarations are added to Ada, Ada exceptions can be specified well enough to allow the verifiation of Ada programs with exceptions. We propose to utilize a variation on [Luckham]'s ideas in a form of structured commentary to assist a reader in understanding a program's use of exceptions.

Commentary on exceptions is potentially needed at four points in a program:

1. declaration of an exception,
2. raising of an exception,
3. handling of an exception, and
4. propagation of an exception.

[Luckham] observes that no formal commentary is needed at the declaration or explicit raising of an exception. We feel, however, that if the exception name, or assertion at the point of raising, is not sufficiently descriptive, additional documentation is desirable. A reasonable layout is:

```
E: exception;              -- why does this exception exist?
...
begin
   ...
   raise E;                -- why is E raised?
```

Associated with each handler must be an assertion which will be true when that handler is executed. A reasonable layout is

```
begin
...
exception
   ...
   when E =>               -- why are we here?
      ...
end;
```

Associated with the specification of each procedure which can propagate an exception must be a set of statements naming each exception which might be propagated, and under what circumstances. A reasonable layout is

```
...
procedure P(...);       -- propagates E sometimes
                        -- propagates E1 other times
                        -- propagates others occassionally
```

This form of commentary, in conjunction with the coding style suggested, will make exceptions in a program easier to understand.

EXAMPLES:

Result Classification

[Course] presents two examples of implementation of result classification reporting. Package RECORD_HANDLER defines an interface to a simple file system; the procedure GET_VALID_RECORD calls GET_NEXT_RECORD in the package body which in turn calls GET within TEXT_IO. If an attempt is made by GET to read past end of file, the exception END_ERROR is raised. Figure 6 shows an implementation in which GET_VALID_RECORD transforms the END_ERROR exception, which is meaningful only to a user of TEXT_IO, into the NO_MORE_RECORDS exception, which is meaningful to the user of RECORD_HANDLER.

Figure 7 shows the implementation of RECORD_HANDLER in which result classification is done by using a status parameter, rather than by raising an exception. GET_VALID_RECORD has a second parameter, END_OF_DATA. It must still handle the END_ERROR exception which it does by appropriately setting END_OF_DATA.

Observe that the END_ERROR exception allows interaction between the different levels of abstraction, GET and GET_VALID_RECORD, while bypassing the intervening GET_NEXT_RECORD level. An alternative implementation would have GET_NEXT_RECORD test the value of the END_OF_FILE function and, when TRUE, pass this status result back to GET_VALID_RECORD, avoiding the use of exceptions entirely.

```
package RECORD_HANDLER is
   type ITEM_RECORD is ...;
   procedure OPEN_FILES;
   procedure CLOSE_FILES;
   procedure GET_VALID_RECORD (REC: out ITEM_RECORD);
      -- propagates NO_MORE_RECORDS when file is empty
   procedure WRITE_RECORD(REC: in ITEM_RECORD);
   NO_MORE_RECORDS: exception;
      -- is raised by GET_VALID_RECORD
      -- when the end of the input file is encountered.
end RECORD_HANDLER;
```

Figure 6 (First Part)

```
with TEXT_IO;
package body RECORD_HANDLER is
use TEXT_IO;
...
procedure GET_VALID_RECORD (REC: out ITEM_RECORD) is
   S: RECORD_STRING;
   LENGTH_ERROR: BOOLEAN;
begin
   loop
      GET_NEXT_RECORD (S, LENGTH_ERROR);
      if LENGTH_ERROR or else not VALID_RECORD(S) then
         WRITE_ERROR_MESSAGE(S);
      else
         REC := CONVERT (S);
         exit;
      end if;
   end loop;
   -- exit from loop occurs only when good record found
   -- or when an END_ERROR exception occurs in
   -- GET_NEXT_RECORD
exception
   when END_ERROR =>  -- GET in GET_NEXT RECORD failed
      raise NO_MORE_RECORDS;
end GET_VALID_RECORD;
end RECORD_HANDLER;


with RECORD_HANDLER;
procedure PROCESS_RECORDS is
   use RECORD_HANDLER;
   ITEM: ITEM_RECORD; -- defined in RECORD_HANDLER
begin
   OPEN_FILES;
   loop
      GET_VALID_RECORD(ITEM);
      WRITE_RECORD (ITEM);
   end loop;
exception
   when NO_MORE_RECORDS =>  -- GET_VALID_RECORD couldn't
      CLOSE_FILES;
end PROCESS_RECORDS;
```

Figure 6 (Continued)

```
package RECORD_HANDLER is
   type ITEM_RECORD is ...;
   procedure OPEN_FILES;
   procedure CLOSE_FILES;
   procedure GET_VALID_RECORD (REC: out ITEM_RECORD;
                               END_OF_DATA: out BOOLEAN);
   procedure WRITE_RECORD(REC: in ITEM_RECORD);
end RECORD_HANDLER;


with TEXT_IO;
package body RECORD_HANDLER is
use TEXT-IO;
...
procedure GET_VALID_RECORD (REC: out ITEM_RECORD;
                            END_OF_DATA: out BOOLEAN) is
   S: RECORD_STRING;
   LENGTH_ERROR: BOOLEAN;
begin
   loop
      GET_NEXT_RECORD (S, LENGTH_ERROR);
      if LENGTH_ERROR or else not VALID_RECORD(S) then
         WRITE_ERROR_MESSAGE(S);
      else
         REC := CONVERT (S);
         exit;
      end if;
   end loop;
   -- exit from loop only occurs when good record found
   -- or when an END_ERROR exception occurs in
   -- GET_NEXT_RECORD
   END_OF_DATA := FALSE;
exception
   when END_ERROR =>  -- GET in GET_NEXT_RECORD failed
      END_OF_DATA := TRUE;
end GET_VALID_RECORD;
end RECORD_HANDLER;
```

Figure 7 (First Part)

```
with RECORD_HANDLER;
procedure PROCESS_RECORDS is
   use RECORD_HANDLER;
   ITEM: ITEM_RECORD; -- defined in RECORD_HANDLER
   NO_MORE_RECORDS: BOOLEAN;
begin
   OPEN_FILES;
   loop
      GET_VALID_RECORD(ITEM,NO_MORE_RECORDS);
      exit when NO_MORE_RECORDS;
      WRITE_RECORD(ITEM);
   end loop;
   CLOSE_FILES;
end PROCESS_RECORDS;
```

Figure 7 (continued)

Queues

In [Habermann] an example is given of the use of generic packages in the generation of isolated abstract objects that are not used in conjunction with one another. The specific example chosen is a queue of complex numbers. This example should, in fact, be extended to include appropriate exceptions for queue overflow and queue underflow, as follows

```
generic
   QSIZE: INTEGER range 1..64;
   type T is private;
package QUE is
   OVERFLOW: exception;          -- attempt to ENQ a full queue
   UNDERFLOW: exception;         -- attempt to DEQ an empty queue
   procedure ENQ (ITEM: in T); -- propagates OVERFLOW
                                 -- when Queue is full
   function DEQ return T;        -- propagates UNDERFLOW
                                 -- when Queue is empty
end QUE


package body QUE is
   FRONT,SIZE: INTEGER range 0 .. QSIZE := 0;
   QBODY: array (1..QSIZE) of T;

   procedure   ENQ  is   ...  end ENQ;
   function    DEQ  is   ...  end DEQ;

end QUE;
```

If a user wants to create a queue of a particular size for a particular type of elements (for complex numbers for instance), he writes in his program the declaration:

```
package PlexQue is new QUE(QSIZE => 36, T => Complex.pair);
```

There may be many similar declarations in a program that each introduce a new queue. Operations on the example queue are denoted by "PlexQue.ENQ(u)" and "PlexQue.DEQ", where "u" is a variable or expression of type Complex. A similar example is found in [LRM] 12.4.

Matrices

Package MATRIX_OPS provides a collection of matrix manipulation facilities, including an example of the recommended documentation for exceptions.

```
package MATRIX_OPS is

   type MATRIX is array (INTEGER range <>, INTEGER <> ) of FLOAT;
   SIZE_ERROR: exception;  -- two matrices are not compatible

   function "+" (M1, M1: MATRIX) return MATRIX;
      -- may raise exception SIZE_ERROR if M1 and M2
      -- are not the same size

   function "*" (M1, M2: MATRIX) return MATRIX;
      -- may raise exception SIZE_ERROR if the number
      -- of columns of M1 is not equal to the number
      -- of rows of M2

end MATRIX_OPS;
```

A use of the MATRIX_OPS package might be

```
declare
   use MATRIX_OPS;
   A,B: MATRIX (1..10, 1..20);
   ...
begin
   ...
   C := A*B;  -- may cause SIZE_ERROR
   E := ...;
end;
```

This block does not have a local handler. Should SIZE_ERROR be raised, it will be propagated to the enclosing unit.

```
package body MATRIX_OPS is

   function "+" (M1, M2: MATRIX) return MATRIX is
      -- may raise exception SIZE_ERROR
      TEMP: MATRIX(M1'FIRST..M1'LAST, M1'FIRST(2)..M1'LAST(2));
      IOFFSET, JOFFSET: INTEGER;
   begin
      if M1'LENGTH(1) /= M2'LENGTH(1) or
         M1'LENGTH(2) /= M2'LENGTH(2) then
         raise SIZE_ERROR;
      end if;

      IOFFSET := M2'FIRST(1) - M1'FIRST(1);
      JOFFSET := M2'FIRST(2) - M1'FIRST(2);

      for I in M1'FIRST(1) .. M1'LAST(1) loop
         for J in M1'FIRST(2) .. M1'LAST(2) loop
            TEMP(I,J) := M(I,J) + M2(I + IOFFSET, J + JOFFSET);
      end loop;
      return TEMP;
   end "+";

   function "*" (M1, M2: MATRIX) return MATRIX is

      -- may raise exception SIZE_ERROR
      TEMP: MATRIX(M1'FIRST(1)..M1'LAST(1),M2'FIRST(2)..M2'LAST(2));
      OFFSET: constant INTEGER := M2'FIRST(1) - M1'FIRST(2);
   begin
      if M1'LENGTH(2) /= M2'LENGTH(1) then
         raise SIZE_ERROR;
      end if;
      for I in M1'FIRST(1)..M1'LAST(1) loop
         for J in M2'FIRST(2)..M2'LAST(2) loop
            TEMP(I,J) := 0.0;
            for K in M1'FIRST(2)..M1'LAST(2) loop
               TEMP(I,J) := TEMP(I,J) + M1(I,K) * M2(K + OFFSET, J);
            end loop;
         end loop;
      end loop;
      return TEMP;
   end "*";

end MATRIX_OPS;
```

File Copy

The discussion of exceptions provided with the input/output packages presented a file copy example. A reasonable approach to this problem is the inclusion of COPY within a package of IO_UTILITIES, as follows:

```
package IO_UTILITIES is
   ...
   IO_UTILITY_ERROR: exception;          -- raised for any exception in any
   ...                                   -- procedure in this package
   ...
   procedure COPY(F: IN_FILE; G:OUT_FILE); -- may raise IO_UTILITY_ERROR
   ...
end IO_UTILITIES;


package body IO_UTILITIES is
   ...
   procedure COPY(F:IN_FILE; G:OUT_FILE) is
      -- may raise IO_UTILITY_ERROR
      C:CHARACTER;
   begin
      while not END_OF_FILE(F) loop
         GET(F,C);
         PUT(G,C);
      end loop;
      return;
   exception
      when others =>
         raise IO_UTILITY_ERROR;
   end COPY;
   ...
end IO_UTILITIES;
```

BIBLIOGRAPHY

[Course] Ada - a Model Course, developed by Georgia Tech for DARPA.

[Goodenough] Goodenough, John B., Exception Handling: Issues and a Proposed Notation, Comm. ACM 18, 12 (Dec 1975), 683-696.

[Habermann] Habermann, A.N., The Use of Ada Packages, Carnegie Mellon University, November, 1980.

[LRM] Reference Manual for the Ada Programming Language, MIL-STD-1815, United States Department of Defense, July 1980.

[Luckham] Luckham, D. C. and Polak, W., Ada Exception Handling: An Axiomatic Approach, ACM Trans. Program. Lang. System. 2,2 (April 1980) 225-233.

[Rationale] Rationale for the Design of the Green Programming Language. Honeywell, Inc. March 1979.

[Schuman] Schuman, S., Tutorial on Ada Tasking, CADD-8104-1601, Massachusetts Computer Associates, Inc., March 1981.

[Steelman] Department of Defense Requirements for High Order Computer Programming Languages, "Steelman", June 1978.

LOW LEVEL LANGUAGE FEATURES

by

Dewayne Perry

Table of Contents

1. Low Level Language Features

This section describes Ada's low level language features and provides examples to illustrate how they might be used. In Section 1.1, we discuss the problems of machine and compiler implementation dependencies, indicating when portability is possible and when it is not. Section 1.2 contains an example of memory oriented hardware interfaces, without interrupts. Section 1.3 extends this example by introducing interrupts and how they are handled in Ada. The low level I/O features are introduced in Section 1.4 and the example is rewritten to use these features of Ada.

1.1. Machine and Implementation Dependencies

In the construction of systems there are particular points where software must be written to interface with hardware, e.g., in managing the use of the processor or peripheral devices. Historically, assembly language has been used at these points instead of higher level languages. These programs written in assembly language are provided in one of two ways, either as part of the operating system that is used by the language or as part of the run-time support system for the language.

In Ada, some of these points of interface (such as processor management) will be supplied by the run-time support. The remaining points of interface are left to the system builder to design and implement. Ada provides several ways of programming this interface. The system builder may use the actual hardware interface or a slightly higher level of abstraction provided by the language or a mixture of both. The choice is more or less determined by the machine with which to interface.

We discuss two major aspects of this interface that must be programmed: device interfaces and instruction interfaces. We show where these interfaces are supported by the language and where they require a higher level of abstraction.

Where the hardware interface takes the form of dedicated memory locations or densely packed data records, Ada provides language features to explicitly describe the interface. Here the designer uses that hardware interface directly and describes dependencies by specifying memory addresses and record representations.

Actual instructions to devices may take either the form of memory references, as in the PDP-11 for example, or, more typically the form of privileged instructions executed by the processor. In the case of memory references, the Ada assignment statement provides the means of device instruction. However, in the case of privileged instructions a slightly higher level of abstraction is required so that the designer does not have to use the privileged instructions directly.

To this end, Ada provides a low-level I/O abstraction embodied in the LOW_LEVEL_IO package.

Two procedures, SEND_CONTROL and RECEIVE_CONTROL, replace the use of privileged instructions which constitute the actual hardware interface. The data interface is also included in this package as part of the encapsulation of the low-level interface. However, the system builder need not be concerned with the data representation since it is hidden within the abstraction.

While the language features that deal directly with the hardware interface are part of the language specification and, as such, are compiler independent, the LOW_LEVEL_IO package is not part of the language specification and is dependent upon the particular implementation. The Ada manual provides a template which suggests the interface in very broad terms (i.e., that there are two procedures and various data structures), but there is no standard for mapping the actual machine interface onto these procedures and data structures. Thus, the low-level I/O interface is susceptible to extreme dependence upon particular compiler implementations.

The system designer is therfore confronted with two types of dependencies: machine interface dependencies and compiler implementation dependencies. These dependencies affect the implementors efforts to make the software portable to other systems.

In the case of machine dependencies, we are confronted with two levels of dependencies: the device interface and the processor interface. The device interface is the set of interfaces such as data and commands that interact directly with the device. The processor interface is the set of facilities that may be required to actually control devices. In other words, the system designer may have to use the processor interface because he cannot directly interact with the device.

Typically, each manufacturer supplies a device interface for any given device that is sufficiently different from that of other manufacturers. It is not possible, therefore to write a generalized low-level device handler. Thus, the designer is forced to rewrite the device handler for each manufacturers device.

Even if device interfaces were identical across several manufacturers, processor interfaces afre different for each manufacturer (and may even be different for machines from the same manufacturer) Thus, certain aspects of the low-level software may be portable while certain other aspects may not be. For example, the use of the processor interface may not be portable. Where the processor interface is directly available to the user and is covered by the language features of Ada, such as address specifications, data representation specifications, memory references and assignment, the software will not be portable.

The compiler implementation dependent low-level I/O package has the potential to help resolve the processor interface problems. Since the low-level I/O package provides a higher level of abstraction than the bare machine, the package could abstract the similarities and hide the differences in the

processor interfaces. If this abstraction occurs, then portability becomes feasible. However, the way the package is implemented could complicate matters rather than simplifying them. Instead of hiding the differences between various device and processor interfaces, the low-level I/O package would compound the differences if the package interface specification were different for each compiler. This would reduce the possibility of writing portable software, even where the software should be portable, as for example on identical machines with identical devices but different compilers.

Thus, while Ada provides a means of making the hardware interfaces visible, it does not provide a sufficient level of abstraction to hide the differences of these interfaces over a large class of machines and devices. Portable low-level software, as a result, will be extremely difficult to write.

1.2. Memory Oriented Interfaces

Data interfaces provided by hardware are typically packed as densely as possible within the processor's unit of memory reference. To map the representation of the high level description of all data interface onto the actual hardware layout, Ada provides record representation specification facilities. To illustrate this we present a simplified card reader handler. The interface described here is similar to that found in a PDP-11: a memory oriented interface. For reasons of simplicity, we will use the status mechanism rather than the interrupt mechanism.

The handler illustrated here will be presented independently of such considerations as gaining permission to use the card reader or viewing the card reader as a virtual resource (see the section on tasking).

Again, for purposes of simplification, assume the existence of a generic package, CIRCULAR_BUFFER_MODULE, that provides an abstract data type, Circular_Buffer, with the operations Get and Put. Assuming that the buffer will only be used by a single producer (the card reader handler), and a single consumer (whoever is using the card reader), we need not concern ourselves with synchronization or exclusion problems while we concentrate on the low level aspects at hand.

```
generic type data is private;
package CIRCULAR_BUFFER_MODULE is
        typeCircular_Buffer (i:natural) is private;
        procedure Get (cb : Circular_Buffer; d : out data);
        procedure Put (cb : Circular_Buffer; d ; data);
private
        type Circular_Buffer ( i : natural ) is . . . ;
end CIRCULAR_BUFFER_MODULE;
```

Essential to the card reader handler are (1) the Card Reader Status Register, a register that

contains the current status of the device and provides the possible commands to the devices; and (2) the Card Reader Buffer Register, a register to specify the address of the buffer for input from the card reader. These are hard wired registers located in what is often called "direct access memory". The registers are first described as types, objects are declared for the types and the representations and locations are specified.

The handler itself is a task that loops forever. Inside the loop, it waits for the reader to become "on-line", the command is issued to read a card, and the handler waits for the card to be read. After completion of the read, either errors encountered are processed or the card is "put" onto the circular buffer. The package interface and the outline of the package body are introduced first.

```
package CARD_READER is
        type CARD_Buffer is new string(1 . . 80);
        procedure Get_Card (b : out CARD_buffer);
end CARD_READER;

package body CARD_READER mis
        CARD_CB_Module is new
                CIRCULAR_BUFFER_MODULE(CARD_Buffer);
                --instantiate a circular buffer for cards
        use CARD_CB_Buffer;
        CB : Circular_Buffer (50);
                -- a circular buffer of 50 cards
        task CR_Handler;
        procedure Get_Card ... end;
        task body CR_Handler ... end;
end CARD_READER;
```

The primary interface procedure is Get_Card which retrieves the oldest card from the circular buffer.

```
procedure Get_CARD (b : out CARD_Buffer) is
        begin
                Get(CB, b);
        end Get_CARD;
```

The task body for the Card Reader Handler contains the type definitions that are appropriate to the device interface. Objects are declared for those device dependant types. Because there is a separation of the logical and physical specifications, the representation specifications occur as a group after the logical definitions. The representation of the data structures and the address specifications of the appropriate device registers are given.

```
task body CR_Handler
-- type definitions
        type CR_Command is (read, . . . );
```

```
        type CR_Error is (timing_error, motion_check, hopper_check);
        type Int_Status is (disabled, enabled);
        type CR_Status is record
                error           : boolean;
                card_done       : boolean;
                problem         : CR_Error;
                on_line         : boolean;
                busy            : boolean;
                ready           : boolean;
                interrupt_status : Int_Status;
                command         : CR_command;
        end record;
        type CRB_pointer is access CARD_Buffer;

-- data object declarations
        CRS : CR_Status;
        CRB : CRB_pointer := new CARD_buffer ((1..80) => ' '));

-- representation specifications for types
        for CR_Command use (read => 1, eject => 2);
        for CR_Error use (timing_error => 1, motion_check => 2, hopper_check => 4);
        for CR_Status use record
                error           at 0 range 0;   -- assume msbit
                card_done       at 0 range 1;
                problem         at 0 range 2..4;
                on_line         at 0 range 5;
                busy            at 0 range 6;
                ready           at 0 range 7;   -- 8 not used
                interrupt_status at 0 range 9;  -- 10-13 not used
                command         at 0 range 14..15;
        end record;
        for CR_Status'SIZE use 16;
        for CARD_Buffer use packing;

--address specifications for data objects
        for CRS use at 8#777160#;
        for CRB use at 8#777162#;

begin -- body of task for handling the card reader
        CRS.interrupt_status := disabled ;  -- status driven handler
        loop
                while not CRS.on_line or CRS.busy
                        loop null; endloop;
                CRS.command := read;
                while CRS.card_done and CRS.busy
                        loop null; endloop;
                if CRS error
                        then log_error (CRS.problem);
                        else put (CB, CRB.data);
                end if;
        end loop;
end CR_Handler;
```

1.3. Interrupts

The status approach, accomplished by constantly polling the device until the desired state is reached, may not be very useful because of the time spent in a "busy wait". An interrupt mechanism enables the handler to respond to changes in device state without busy-waiting.

The task entry mechanism is used to provide the means of specifying interrupt connections and interrupt enabling and disabling. By representing an entry at a particular location (probably an interrupt vector location) the system designer connects the entry to the actual interrupt. When an **accept** statement is performed on the entry, the interrupt is enabled until the rendezvous actually takes place: otherwise the interrupt is disabled. This requires the runtime support to put a layer of abstraction between the Handler and the interrupt mechanism.

We present here only the card read handler. The remainder of the package stands unchanged. The interface, except for the use of interrupts, is identical.

```
Task CR_Handler is
        Entry CR_Int;
        for CR_Int use at 8#100#;
end CR_Handler;

task body CR_Handler is
begin
        loop
                CRS.command := read;
                accept CR_Int;
                if CRS.error
                        then log(CRS.problem);
                        else put(CB, CRB.data);
                end if;
        end loop;
endCR_Handler;
```

Notice that the interrupt handler does not interract with any other process except the hardware process through the entry (as an interrupt call) mechanism. It does not perform a call on the entry of any other process nor does it **accept** calls from other processes. Extreme care must be taken to insure that the handler does not wait unknowingly on another process, especially if that wait degrades the efficiency of the handler beyond acceptable levels. This wait could easily occur if the handler calls the process that was suspended to allow the handler to service the interrupt.

1.4. Low Level I/O Package

Where the device dependant interface is provided by means of priviledged processor instructions, the system builder must use the implementation dependant interface provided by the low level I/O package. The allowed commands for each device will be specified in the parameters for the SEND_CONTROL procedure. The device handler instructs the desired device through this procedure. The status retreivable from the device is defined in the second parameter of the RECEIVE_CONTROL procedure. By calling this procedure, the current state of any device may be determined. The actual code for the handler is similar to that found in the previous two sections, except that most of the interface mechanism is exported by the low level I/O package.

We present her only the part of the package that is relevant to the card reader handler.

```
package Low_Level_IO is
        type Device_Class is ( . . . . cardreader, . . . );
        type CARD_Buffer is new string(1 . . 80);
        type CR_Command is record
                c : (read, eject);
                d : CARD_Buffer;
        end record;
        type CR_Error is (timing_error, motion_check, hopper_check);
        type CR_Status is record
                error           : boolean;
                card_done       : boolean;
                problem         : CR_Error;
                on_line         : boolean;
                busy            : boolean;
                ready           : boolean;
        end record;
        . . .
        procedure SEND_CONTROL (d : Device_Class; c : CR_Command);
        procedure RECEIVE_CONTROL (d : Device_Class; s : CR_Status);
        . . .
end Low_Level_IO;
```

The task interface remains the same as above. The entry connects the handler to the interrupt. The body of the task has calls to the low level I/O routines instead of the memory references that sufficed in the previous examples.

```
Task CR_Handler is
        Entry CR_Int;
private
        for CR_Int use at 8 # 100 # ;
end CR_Handler;

task body CR_Handler is
        use Low_Level_IO;
```

```
        CRS : CR_Status;
        CARD : CARD_Buffer;
begin
        loop
                SEND_CONTROL (cardreader, (read, CARD));
                accept CR_Int;
                RECEIVE_CONTROL (cardreader, CRS);
                if CRS.error
                        then log(CRS.problem);
                        else put(CB, CARD);
                end if;
        end loop;
end CR_Handler;
```

1.5. Summary

We have illustrated how a system builder might construct machine dependant software in various kinds of machine architectures. Not all of the language features that are available to express machine dependancies have been used in these examples, but the ways in which they might be used are analagous to what has been shown.

Particular care must be taken if portability is a major issue. Two types of dependancies must be reckoned with: hardware and compiler implementation dependancies. These issues cannot be avoided, but the designer can, by suitable abstractions, confine these sections of non-portable code and, thus, minimize their impact when porting software from one system to another.

In general, the features of Ada enables hardware dependancies to be made visible at the language level and provides a means of logical treatment that attains all the benefits of readability and maintainability that one expects from high level languages.

AD . . . . .

TUTORIAL MATERIAL ON THE REAL DATA-TYPES IN ADA

Final Technical Report

by

B A Wichmann

November 1980

United States Army

EUROPEAN RESEARCH OFFICE
London, England

CONTRACT NUMBER DAJA37-80-M-0342

National Physical Laboratory
Teddington, Middlesex, TW11 0LW, UK

133

AD

TUTORIAL MATERIAL ON THE REAL DATA-TYPES IN ADA

Final Technical Report

by

B A Wichmann

November 1980

United States Army

EUROPEAN RESEARCH OFFICE
London, England

CONTRACT NUMBER DAJA37-80-M-0342

National Physical Laboratory
Teddington, Middlesex, TW11 OLW, UK

134

Abstract. The Ada programming language introduces a number of novel features in the area of numerics. The purpose of this report is to present these features to a programmer who is familar with numerical computation but not with Ada. The report is designed to be presented as a lecture with the aid of viewgraphs (drafts of these are included). However, the material can be read in the conventional fashion.

Comments. The author would appreciate comments on the text so that any subsequent revision can incorporate improvements.

Acknowledgment. Mr G T Anthony and Miss H M Williams of NPL have reviewed this report which has resulted in substantial improvements in its style and content.

CONTENTS

Tutorial material on the real data-types in Ada

B A Wichmann, National Physical Laboratory

Note: These notes are designed as material to be presented with a set of viewgraphs. The complete material can be presented in about four hours assuming only a limited knowledge of Ada beforehand. The viewgraphs are reproduced at the end of these notes, and are referenced in the text by numbers in the right hand margin.

(1)

1. Fixed and Floating point

The real data types in Ada are for approximate computation. The majority of physical quantities are necessarily approximate because of the inherent errors involved in their observation. Such quantities are therefore naturally handled by means of the real data types in Ada. The purpose of these notes is to explain the facilities in Ada so that the programmer can use the language reliably and in a manner appropriate to the job in hand.

The real data types in Ada are divided into two classes - fixed point and floating point. There can be any number of fixed point and floating point data types in a program. It is convenient to have an intuitive view as to what fixed point and floating point means. Thinking in decimal, fixed point means a fixed number of places before the decimal point and a fixed number after:

±d.dd or ±ddd.d or ±ddd.

whereas floating point means that there are a fixed number of significant digits and an exponent ("scientific notation" of calculators): (2)

±d.ddE±dd or ±d.dE±d or ±d.dddE±dd

where the integer after the E gives the decimal exponent.

The data type thus determines how values are stored since any one type will have the same format. With a fixed point data type with the format

±d.dd

a half is stored as +0.50 and one third as +0.33 which is, of course, in error to a small extent. The fact that computed values and even constants cannot be stored exactly is the reason why real

data types are said to be approximate. Note that with this data type values of magnitude less than 0.005 will be represented as zero (assuming rounding is performed).

Now consider an example of a floating point data type with the format: (3)

±d.ddE±d

Then

100.0 is stored as +1.00E+2

One might think it could also be stored as +0.10E+3 but this is not permitted because values are `normalized'. The importance of normalisation is easy to appreciate when considering storing the value 101.0 with the same format. This is

+1.01E+2

whereas putting an initial zero would lose the final 1 giving a one per cent error. Note that floating point values have a roughly constant relative error whereas fixed point quantities have a constant maximum absolute error.

In Ada, data types are distinguished by their names, not just their formats. Hence two data types having identical formats are distinct. This means that data types should be given names to reflect the logical properties rather than their formats. If two sensors read temperature and distance, then they should be given distinct data types DEGREES and FEET rather that one type just because the range of values and accuracy requires an identical format.

Note that the fixed point data types in Ada have formats which are not quite the same as those used by calculators. With a calculator, dividing 1.23 by ten gives .123, but with the format +d.dd, the division will yield +0.12. To do the division accurately in Ada, then the result must be stored in a type with the format +.ddd. Clearly, the reduced flexibility of the Ada fixed point means that it is very easy to lose accuracy in performing computations. Losses also arise with floating point computations but they are less marked due to the automatic normalization. For this reason, most programmers would prefer to use floating point, which is of course, why modern scientific computations use this mode. As a rough estimate, one should expect an algorithm to be three times more expensive to program in fixed point. The reason for using fixed point is usually the absence of floating point on a particular machine or because the digitised signal input is in fixed point. (4)

The description given above using decimal formats is merely to illustrate the general nature of Ada data types. In fact, Ada

defines the data formats in binary since this is almost universal for modern computers. To give a more accurate description, we first need some notation from the Ada language.

2. Notation for literals

We are only concerned with numeric literals. These can either be for integers or real values. Real values are distinguished by the presence of a decimal point. If a real value is required by a particular context in the language, then an integer is not permitted. In other words, if the real value one is required 1.0 must be written and 1 will not be sufficient. This means that it is always easy to see if approximate computation is being performed, even with parameters to a procedure because literal values will have a decimal point.

Decimal integer values are written in the conventional manner. Spaces may not appear within the digits of the value, but an underscore can be used instead. This is very convenient with large values since the thousands or millions can be separated to aid the eye. (5)

Examples: 1 01_234_567

The following are not valid

1_ 2. 1__234

Large integer values can conveniently use the exponent notation. For instance, six million can be written as:

6_000_000 or 6E6 or 6_000E+3 etc.

An implementation may limit the size of literals which can be handled, but such limits are likely to be quite large. The line length also restricts the magnitude of literals.

Real literals can be written in the conventional decimal notation with a decimal point. An exponent can optionally be used. For instance, the following all represents the same value:

3.14 0.314E+1 314.0E-2 03.1_4000

The accuracy with which a literal value is stored in the program is determined by the context and not by the way in which the literal value is written in the program. Hence merely writing 20 decimal digits does not imply that the value will be stored with that accuracy. The accuracy will depend upon the types used in the computations containing the literal.

Both integer and real literals can be written using bases other than ten. One reason for this facility is that some machine

properties are specified by the manufacturer in octal or hexadecimal and hence this notation is the logical one to use in these contexts. An additional reason for permitting other bases for real literals will soon be apparent. The bases which Ada allows are those from 2 to 16. Base 16 uses a notation similar to that of hexadecimal on the IBM computers and in consequence A stands for 10, B for 11, C for 12, D for 13, E for 14 and F for 15. The base is determined by a decimal value before a sharp character which brackets the based number sequence. Note that the base and the exponent are written in decimal and in consequence the A-F characters when used as a digit, can only appear between the pair of sharp characters. For example:

2#101# means 4 + 1=5 with a base of 2

4#101# means 4^2 + 1 = 17 (6)

16#FF# means 15 * 16 + 15 = 255

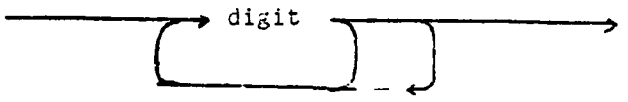
The notation can be used both with exponents and with a point for real literals.

Hence 4#101#E2 means 4#10100# = 4^4 + 4^2 = 256+16 = 272

Writing and reading values in other bases requires care since we tend to think in decimal. This is especially true with real values.

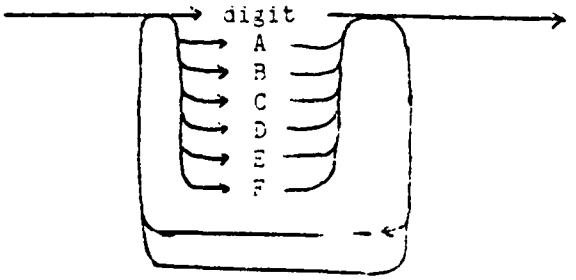
The syntax of numeric literals is most easily portrayed by means of syntax diagrams. The arrowed lines are followed according to the syntax units being analysed. For instance, an integer with interleaved understores permitted is given by the diagram

integer:



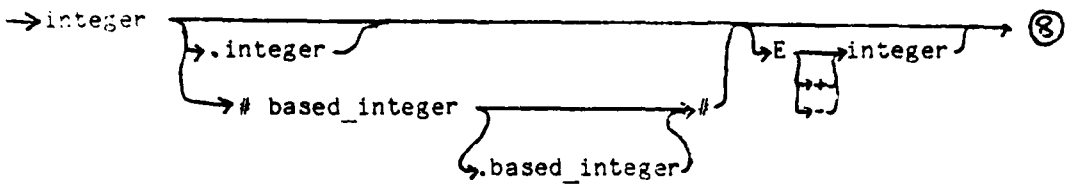
The box for digit can also be given by a diagram with just ten (7) alternatives for each of the digits 0 to 9. Similarly, one has

based_integer:



Now the syntax diagram for numeric literals can be given using the diagrams for integer and based_integer

numeric_literal:



(8)

One further language facility needs to be given because of its convenience in explaining the language later in this material. This is number declarations. Both real and integer literal values can be given an identifier in a number declaration. For instance

```
PI: constant := 3.14159_26535;
MAX_LINE_LENGTH: constant : = 96;
```

Within the Ada program where these identifiers can be used, the use of the identifier is equivalent to writing the literal value. Such number declarations can be used to separate out key numerical values.

Exercises

Write the following based number values in decimal:

16#FF# 4#1.01#E2 3#0.1# 8#0.1# 16#0.8#

(9)

What value is 16#0.99999# just a bit less than?

What is wrong with the following literals?

3._14 4#_0.1#2 16#FF#E-1 8#0.9#

3. A model of approximate computation

Ada defines the properties that the approximate computation of real arithmetic must satisfy. Because the real arithmetic is implemented on machines with very different underlying hardware, the definition is permissive. In other words, the properties must be satisfied, but this can be achieved in a number of different ways. A particular real data type definition specifies an accuracy that must be met. An implementation is free to provide greater accuracy than that specified. This is essential because a machine can usually only conveniently implement a small range of different accuracies. The problem is to define the properties so that

141

different implementations are possible and yet make the properties good enough to meet the demands of the numerical analyst. The method used is based upon the work of W S Brown from Bell (10) Laboratories on floating point [2]. There are differences between Brown's work and the definition of Ada because of the different objectives - Brown was interested in providing a model of actual hardware whereas with Ada a machine independent language definition is required. Ada also handles fixed point.

Ada assumes that the arithmetic facilities are provided using binary. There are a few additional complexities with fixed point, so let us start by considering floating point. Floating point computation involves storing values with a sign, a mantissa and a signed exponent. The difficulty is that we do not wish to say how long the mantissa will be, nor the actual range of the exponent since this will depend upon the particular hardware in use. Hence we say that the mantissa must be at least so long, and the exponent range must be at least so long.

With a particular mantissa length and exponent range guaranteed, certain values are capable of being stored exactly. As an example, assume that the mantissa length is 4 hexadecimal places (corresponding to 16 binary places). Then

```
          16#0.8000#      = 0.5 is stored exactly
as is     16#0.F000#      = 15.0/16  = 0.9375
 and      16#0.FFFF#E4    = #FFFF.0# = 65535.0
```

With such a data type, these values are handled exactly in the sense that if one assigns a value to a variable, then one can test for equality and obtain the expected result. These values are called model numbers. Equality and inequality of model numbers have the characteristics of the exact values. However, an implementation will typically have values which are not model numbers and almost all the difficulties of real arithmetic are due to these values. Given one of these additional numbers it is usually bounded by a model interval. For instance, with the above data type

```
16# 0.ABCD4#      is bounded by
16# 0.ABCD#  and  16#0.ABCE#
```

and 0.1 which is 16#0.1999999...# is therefore bounded by

```
16#0.1999#  and   16#0.199A#
```

Literal values in an Ada program must be converted by the compiler to values within these bounds inclusively. Hence these model intervals perform a vital role in defining the errors that can (11) arise in a computation. This role is extended to operations as follows. Given two operands A and B and an operation op, then we want to bound A op B. Corresponding to A and B, there are model intervals. The operation is then applied to the two intervals. The

resulting set of values is then widened, if necessary, to a further model interval. This model interval bounds the machine computed value of A op B. This might seem complicated and indirect, but it has a number of simple consequences. For instance, the model interval for a model number is just the model number. Hence if the operands are both model numbers and the correct, mathematical result is also a model number, then the machine result must be the exact, correct result. (12)

Exactly the same logic of model numbers, model intervals and the calculation of model intervals which bounds the result of an operation applies to fixed point as well. The difference between fixed point and floating point lies in the model numbers themselves. There are some additional complexities which arise in the case when a computed result lies outside the range of model numbers.

Exercises

Given a floating point type which has model numbers with 4 hexadecimal places, what is

(a) the next model number above 1.0?
(b) the next model number below 1.0?
(c) the ratio of ((a) - 1.0)/(1.0 - (b))?

What rational numbers are not represented exactly in Ada with any accuracy using floating point?

4. Floating Point Data types

Ada allows the programmer to specify the minimal accuracy of a real data type. For floating point this specification is an integer giving the number of decimal digits of significance in ... ed values. This method of specification is used because of its ... ong intuitive appeal in spite of the fact that the detailed semantics of floating point uses binary.

The number of decimal digits determines the model numbers of the type. Since a binary radix is used, the floating point model numbers consist of

```
sign * binary_mantissa * (2.0 ** exponent)
```

(13)

where the mantissa length and the exponent range must be determined from the number of decimal digits. There is an obvious relationship between a binary mantissa and the corresponding decimal one. For D decimal digits one needs more than D*log(10)/log(2) binary places to give at least the same accuracy. Hence Ada defines the mantissa length to be the next integer greater than D*log(10)/log(2).

Unfortunately, there is no obvious natural value for the exponent range. In fact, the exponent range is independent and logically should be separately specified by the programmer. Such a detailed specification would not be useful since actual hardware does not permit an independent choice of both parameters. Also, to be convenient to the ordinary user, default values are needed for the range which would again be arbitrary. Some algorithms do require a reasonable range in relation to the mantissa and from a study of existing machines, the value has been set of -4*B .. 4*B where B is the number of binary places in the mantissa.

Let us now consider an example of a minimal working accuracy of five decimal digits. This requires at least 5 * 3.32 = 16.6 binary places. Hence the length of the binary mantissa is taken as 17 places. The binary exponent range is -68 .. 68. Hence we have

```
smallest model number greater than zero      = 2#0.1#E-68
                       = 2.0**(-69) about 1.69E-21

largest model number   = 2#0.1111111111111111#E68

                       = 2.0**68 - 2.0**51 about 2.95E20

The next model number greater than 1.0
                       = 2#0.1000000000000001#E1
                       = 1.0 + 2.0 **(-16)
```

(14)

Such a floating point type is defined by the declaration

```
type F is digits 5;
```

Having declared such a type, it is clearly convenient to be able to access the basic constants associated with it. By this means, algorithms can be written where the accuracy is isolated to the single type declaration. These constants are called attributes of the type and, in this case, they are predefined by the language definition.

The predefined attributes are written as the type identifier (F) a prime (') and then the name of the attribute. The attributes for a floating point type which are related to the model numbers are:

F'DIGITS: the value of the expression after 'digits' in the type declaration, and hence 5 in this case,

F'MANTISSA: the binary length of the mantissa and hence 17 in this case,

F'EMAX: the maximum value of the exponent which is 68 in this case and is always 4*F'MANTISSA

F'SMALL: the smallest positive model number, which is about 1.69E-21 in this case. Its value is

always 2.0**(-F'EMAX-1).

F'LARGE: the largest model number, which is about 2.95E20 in this case. Its value is always 2.0**F'EMAX*(1.0-2.0**(-F'MANTISSA)),

F'EPSILON: the absolute value of the difference between 1.0 and the next model number above 1.0. The value in this case is about 1.52E-5 or in general 2.0**(-F'MANTISSA+1)

Of course, because of the relationship between these values, there is little logical need for them all. In practice, however, they are needed for program clarity. F'MANTISSA and F'EMAX give the basic properties of the model numbers whereas in actual programming the values F'SMALL, F'LARGE and F'EPSILON are usually needed.

Consider the problem of determining the errors in a computation. A literal value such as 0.1 cannot be stored exactly since it has a recurring binary representation. What is the error involved? In handling binary values, it is convenient to use hexadecimal otherwise the based numbers are rather long to write. We have (15)

0.1 = 16#0.19999...#

With the type F we have 17 binary places and hence the value 0.1 is bounded by the model interval

16#0.19999#..16#0.1999A#

The difference is 16#0.00001# = 16#0.1#E-4 = 16.0**(-5)
= F'EPSILON/16

The relative error is thus less than or equal to 9.54 E-6 in this case.

In general, it is easy to see that the relative error depends upon the relationship between the value and the powers of 2. For instance, a value just greater than one has a relative error of F'EPSILON (the definition of the value) whereas a value of just less than 1.0 has half that relative error. In practice, the actual values of constants are not so important, and in any case cannot be used for variables and hence the general rule is

lowest possible machine value representing the true value
= (1.0-F'EPSILON)*true value
highest possible machine value representing the true value
= (1.0+F'EPSILON)*true value

These are the relationships used for classical error analysis, combined of course, with corresponding relationships involving the numerical operations. Note that constants may be converted by

rounding implying half the maximum relative error. This paper does not aim to teach classical error analysis. The formal proof of the inequalities of classical error analysis from the Ada model number definition is given in References [1,2].

Classical error analysis is satisfactory provided the values computed are either zero or lie in the ranges F'SMALL .. F'LARGE and -F'LARGE .. F'SMALL. Consider the computation of a value smaller in magnitude than F'SMALL, or even such a value written in a program. Then the value in the machine will be in interval -F'SMALL .. 0.0 or 0.0 ..F'SMALL according to the sign of the value. This implies that all precision could be lost. For instance, the actual machine may only handle model numbers and round literal values. Hence values greater than 0.0 and less then F'SMALL/2 will be converted to zero. A compiler could warn the programmer of such a conversion of a non-zero value to zero, but there would be little reason to do so since the same values calculated dynamically would lead to zero without warning. Hence the programmer needs to beware of this condition called underflow, if an algorithm requires the accurate computation of small values.

As an example of underflow, consider the computation of the length of the hypotenuse of a right angled triangle:

```
X := SQRT(A**2 + B**2);
```

It might seem reasonable that if A or B >= F'SMALL then X >= F'SMALL. However, F'SMALL**2 may underflow to 0.0, giving X=0.0 if both values are small. Hence if the specification of this calculation requires that non-zero values of A or B gives a non-zero value for X, then one must take this into account by writing (for instance):

```
SM: constant F := 2.0**(-F'EMAX/2);      --EMAX is even
-- calculate A and B
if ABS(A) < SM and ABS(B) < SM then
     A := A/SM;
     B := B/SM;
     X := SQRT(A**2 + B**2) * SM;
elsif
     -- other case
end if; -- (*1)
```

Note that the use of powers of two for scaling reduces the potential errors to a minimum.

Ada does not require that there are no machine values between 0.0 and F'SMALL. On a particular machine, such values could be present making the cautious code above less necessary. The programmer is

---

(*1) This example is merely an illustration, see section 10 for a realistic example.

strongly advised to take the precautions for underflow illustrated above because the algorithm will then be portable.

The problem of overflow, that is, when computed values or constants are greater than F'LARGE, is more severe. Clearly, there must be some limit to values that a machine can handle and beyond that limit it is, in general, unreasonable to replace the true value by a single value. Ada only requires that values upto F'LARGE are handled correctly. A machine can, and often does, provide further values. The implemented range for any Ada scalar type is F'FIRST .. F'LAST. When the implemented range of values is exceeded, most machines provide an indication of this fact. In Ada, this is signalled by means of the NUMERIC_ERROR exception, for computed values. If a literal value exceeds the implemented range, then the CONSTRAINT_ERROR exception is raised. With underflow, the computation proceeds in spite of obtaining potentially meaningless results, but with overflow an exception could lead to the termination of the computation. Hence the specification of a numeric computation should indicate if these exceptions can arise. The specification of a routine should indicate which of the following three cases hold with respect to the NUMERIC_ERROR and CONSTRAINT_ERROR exceptions:

(17)

(18)

(a) The routine has been written so as to avoid raising the exceptions.

(b) Local handlers have been written for the exceptions so that these exceptions cannot be propagated to the caller.

(c) The exceptions can indeed arise from a call of the routine (the conditions should be stated).

Consider now the computation

```
X := SQRT(A**2 + B**2);
```

but this time considering the question of overflow. The safest method is to avoid overflow by testing the values of A and B in a similar method of that used for underflow:

```
SL: constant F := 2.0**(F'EMAX/2-1);
-- calculate A and B
if ABS(A) > SL or ABS(B) > SL then
   A := A/SL;
   B := B/SL;
   X := SQRT(A**2 + B**2) * SL;
elsif
   -- other cases
end if; (*1)
```

An alternative strategy is to write a handler for the NUMERIC_ERROR exception and only in this case, scale for a large value. This is not to be recommended in general because it is machine dependent. The raising of the NUMERIC_ERROR exception is not guaranteed and indeed, on machines which allow computation with values representing infinity, the exception might never be raised.

A user can declare subtypes of a type (or subtype). Unlike a type, a subtype is potentially dynamic in its characteristics. Consider

```
type F is digits 5;
X:F := F(READ_FROM_DEVICE);
subtype TF is range 0.0 ..X;
```

Then the range of values that the subtype TF can have may vary from one execution of these declarations to another. On the other hand, the properties of F remain the same since the expression after `digits' is a static integer expression.

Subtypes of real types have both advantages and disadvantages in Ada. Obviously, it is useful to place bounds on values and have these bounds checked by the system as both a documentation aid and also to improve the reliability of the software. Unfortunately, the checking overhead on every assignment to variables of subtype TF is not insignificant. The check is necessary since the program is required to raise the exception CONSTRAINT_ERROR if the range is violated. The programmer can suppress the checking by means of a pragma, but this defeats the object of the facility. Hence subtypes with a real range constraint must be used with care.

Subtypes can also be used to indicate a need for less accuracy than that specified by the type definition. For instance:

```
subtype SF is F digits 4;
```

or just against an object

(*1) Again, this example is illustrative only, and section 10 gives a realistic example.

```
Y: F digits 4;
```

For a subtype, the model numbers are reduced by a corresponding reduction in the mantissa length, while keeping the exponent range the same. Hence this would mean a binary mantissa length of 14 places (3 less than F). This means that SF'LARGE is only a very small amount less than F'LARGE corresponding to losing three 1's at the least significant end of the binary mantissa. Note that SF'SMALL = F'SMALL. (19)

Since compilers must handle objects of a subtype in effectively the same way as objects of the type, it is unlikely that compilers can take much advantage of the reduced precision of a subtype. Hence the advantages of subtypes just giving an accuracy constraint are minimal. Since there is no checking for accuracy constraints at run-time, there is no run-time penalty.

Exercises

If F'DIGITS = 2*G'DIGITS, does F'MANTISSA = 2*G'MANTISSA? (20)

What is the largest positive value X:F such that X does not overflow and 1.0/X does not underflow?

5. The predefined floating point operations

For every floating point type, a conventional set of predefined operations are available as follows: (21)

single operand	+	no operation
	-	change sign
two operands (of the same floating point type)	*	multiplication
	/	division
	+	addition
	-	subtraction
single parameter	ABS( )	absolute value

Each of these operations yields a result which is of the same type as the operands. The description of the error bounds and the circumstances under which the exception NUMERIC_ERROR can occur can now be given in detail (see 4.5.8 of manual), by means of the following steps:

1. For each operand, a model interval of the appropriate type or subtype is obtained.

2. The mathematical operation is performed on the model intervals, obtaining a new interval.

3. The interval from the last step is expanded, if necessary, to a model interval.

The model interval obtained from this last step bounds the accuracy of the operation.

Consider the computation of X/Y; X,Y:F and X=15.0 and Y=3.0. Both X and Y are model numbers (as are all small integers). Hence the two model intervals obtained from step 1 are just the two single values. Step two yields the mathematical result 5.0. Now step 3 gives the model interval consisting of this single value, since 5.0 is also a model number of type F. One can clearly see from this that computations involving small integer values and giving small integer values are exact.

Consider now a slightly more realistic example of X*Y, X,Y:F and X = 0.1 and Y = 10.0

Then X is in model interval 16#0.19999#..16#0.1999A#
and Y is the model number 16#0.A#E1

Step 2 then gives the interval 16#0.FFFFA#..16#1.00004#
Step 3 then gives the model interval 16#0.FFFF8#..16#1.0001#

If the programmer had written 0.1*Y in his program, then 0.1 is converted to the type F by the compiler and hence the same error analysis applies. Note that the resulting bounds are approximately symmetric about the correct result, although some actual machines may produce results with these bounds but with a bias. (22)

Consider another example of X+Y, X,Y:F with X=1.0 and Y=F'SMALL. Then the interval at step 2 is the single value 1.0 + F'SMALL but this is widened to the model interval 1.0 .. 1.0 + F'EPSILON. This analysis assumes that F'SMALL < F'EPSILON which is a consequence of fixing the exponent range in relation to the mantissa length.

One situation has not been detailed. In steps 1 and 3 above, it may be impossible to form a model interval because a value exceeds F'LARGE in absolute value. In this case, the interval is said to overflow. When this happens, the NUMERIC_ERROR exception may be raised. It need not be raised because the machine can handle larger values adequately or because no indication is given by the hardware. Because these different circumstances cannot be distinguished, portable software cannot rely upon the NUMERIC_ERROR exception.

One other operation is available for floating point which is irregular since the operands are of different types. This is the exponentiation operator written as **. The left hand operand is any floating point type and the right hand operand is any integer type. The result is of the same type as the left hand operand. The operation gives the result of repeatedly multiplying the left hand operand by itself for a positive exponent. The number of

multiplications being one less than the value of the right operand. A negative exponent gives the inverse of the positive expoent value. Hence:

```
X **2      is equivalent to X * X
X **(-2)          "         1.0/(X*X)
X **1             "         X
X ** 0            "         1.0
```

(23)

Hence the semantics of this operation are defined in terms of the multiplications involved. The compiler can reduce the number of multiplications by calculating X**4 as (X*X)*(X*X) rather than (X*X)*X)*X. This gives a faster computation for large values of the exponent but does not give (in general) more accuracy.

The remaining operations on floating point operands are more regular than ** but give a BOOLEAN result. These are the relational operators. All six relational operators are available although they must be used with caution, as we shall see.

In comparing two values, everything is straightforward if the two values are not approximately equal and both are in range (ie between - F'LARGE and F'LARGE). However, if the two values are nearly equal, one has a potential problem. Under such circumstances, the result will depend upon the actual accuracy of the hardware. The precise formulation of this again depends upon the use of model intervals as follows:

Firstly, the appropriate model intervals are constructed for each operand as in the case of the other operations. Then one of five cases determines the result:

(a) The intervals are disjoint: the mathematical result is obtained
(b) Each interval is the same single model number: the mathematical result is obtained (24)
(c) The two intervals intersect in a single model number: either the exact result is obtained or that of comparing one operand with itself
(d) The intervals have more than one number in common: the result is implementation dependent.
(e) One of the two intervals overflows: the result is again implementation dependent, but the NUMERIC_ERROR exception can be raised (although it need not).

These cases are easily illustrated by means of a table with type F of five digits again.

X	op	Y	case	result
0.1		10.1	(a)	mathematical result
F'SMALL		F'SMALL	(b)	mathematical result
0.1		0.1+F'EPSILON/8	(c)	(Intersect at 16#0.1999A#) mathematical result or 0.1 op 0.1 (=Y op Y)
0.1		0.1+F'SMALL	(d)	implementation defined
F'LARGE		-F'LARGE+1.0	(e)	implementation defined or NUMERIC_ERROR
0.1		0.1	(d)	implementation defined (*1)

Exercises

With A, B, C:FLOAT;

(25)

Does

```
(A + B) + C = A + (B + C)?
A + B = B + A?
```

What is wrong with the following?

```
A + 12
24*B
C**2.0?
```

6. Derivation from the hardware types for floating point

As explained so far, it would appear that an implementation would have to provide a large number of distinct types for digits N, N=1..30 (say). However, as is well known, machines typically have only one or two hardware types. We would appear to have a problem. However, as defined, an accuracy of N digits can be implemented with a hardware type having N or more digits of accuracy. Hence, given a machine with two hardware types of 10 and 20 digits accuracy, all the types of accuracy <= 10 would be handled with 10 digits, and the ones with more than 10 and less than or equal to

---

(*1) It might seem odd that 0.1=0.1 is not necessarily true. The reason is that many machines perform calculations with more accuracy than results can be stored in main memory. This is the so-called overlength accumulator. Hence 1.0/10.0 would give more accuracy than 0.1 stored in main memory, giving the unexpected false to 1.0/10.0 = 0.1.

20 digits with 20 digits of accuracy. The vital fact which permits this is that the model numbers for accuracy of digits N are model numbers for all larger accuracies. (26)

The hardware types have conventional names, namely SHORT_FLOAT, FLOAT and LONG_FLOAT. Of course, if there are only two hardware types, the names actually in use will depend upon the implementation. A valid Ada system could have no such types if the target hardware provides no approximate facilities. Assuming that floating point is provided, then type FLOAT should be available. Hence, if one is not concerned with control of accuracy for small amounts of code, then one can just use the type FLOAT. Direct use of the hardware types is not to be recommended since it is clearly machine dependent. However, if it is necessary to implement basic software effectively to augment the hardware, then such machine dependence is needed. Note that the attributes of FLOAT (SHORT_FLOAT and LONG_FLOAT) characterise the machine.

Given

```
type F is digits D;
```

such that F is implemented by the hardware type FLOAT, we say that F is derived from FLOAT which is written in full as

```
type F is new FLOAT digits D;
```

The full form is not appropriate in most cases, since on another system F could be implemented by SHORT_FLOAT. Hence the short form of declaration is to be preferred to increase portability. Even with the long form, F'DIGITS = D and this is not necessarily equal to FLOAT'DIGITS. Given either form of declaration, it is occasionally necessary to access the characteristics of the implemented type. This can be done by means of the notation (27)

```
F'BASE'DIGITS meaning FLOAT'DIGITS etc.
```

The advantage of the 'BASE notation is that it is possible to exploit the additional fortuitous accuracy provided by the implementation. Consider for instance the summing of a series T(I) until convergence is obtained:

```
SUM := 0.0;
while ABS(T(I)) > F'EPSILON * SUM loop
   SUM := SUM + T(I);
   I := I + I;
end loop;
```

As written it will stop summation appropriate to the declared properties of F. However, on a particular machine more accuracy might be obtained by writing F'BASE'EPSILON - going further than that would be pointless. Of course, a numerical analyst would sum such a series from the smallest term upwards, but the principle

remains the same.

Exercises

What is the relationship between F'DIGITS and F'BASE'DIGITS?

What is the relationship between F'LARGE and F'BASE'LARGE?

If F'DIGITS = G'DIGITS does F'BASE'DIGITS = G'BASE'DIGITS?

7. Fixed point data types

With fixed point data types, the user specifies the maximum acceptable absolute error bound. It is also necessary to specify the total range of values that must be covered, since the range and the error bound are required to determine the representation of values. A fixed point data type has the form:

```
type FX is delta D range L .. U;
```

where the D, L and U are static real expressions. All three values can be accessed as attributes of the fixed point type:

D = FX'DELTA, the absolute error bound,
L = FX'FIRST, smallest value of the type,
U = FX'LAST, the largest value of the type.

The type definition, together with a possible representation specification determines the set of model numbers of the type. The model numbers of the type are integer multiples of a value called the actual delta, which is an attribute of the type (=FX'ACTUAL_DELTA). This value is smaller than or equal to FX'DELTA so that values can be represented to within the accuracy specified. The range of integer values for the model numbers must be sufficient to be within FX'DELTA of both L and U. In an analogous way to the mantissa for floating point, the integer multiple (including the sign) is assumed to have a total range of -2**N+1.. 2**N-1 for some N. Hence to summarise, the model numbers are: (23)

```
sign * multiple * FX'ACTUAL_DELTA
```

where 0 <= multiple <= 2 ** N - 1 (for some N).

There are some essential differences here between fixed point and floating point. In floating point, some values such as 1.0 are always model numbers. This is not the case with fixed point. The value 1.0 could be less than FX'ACTUAL_DELTA and in consequence represented as 0.0. Conversely, 1.0 could exceed the range of values of the type and hence use of such a value could raise CONSTRAINT_ERROR. In general, unless a representation specification has been given which explicitly states the value of

ACTUAL_DELTA, the model numbers are unknown. Hence one cannot assume that certain values, such as powers of two which are in range, will be represented exactly.

Let us now consider a practical example. The need is to process data which has an observational error of 0.01 and a range of 0.0 to 100.0. To ensure the ability to hold negative values, the type definition could be

```
type F is delta 0.01 range -100.0 .. 100.0;
```

A typical implementation could then choose a power of two for the ACTUAL_DELTA. This could be 1.0/128 or a smaller power, depending upon the word length of the machine. In this case, assume that (29)
F'ACTUAL_DELTA = 1.0/128. Then the model numbers are multiples of this value to a limit which must be at least within the range -100.0 + 0.01 .. 100.0 - 0.01. Since the multiples are a power of two, the model numbers are:

```
M * 1.0/128 where -2**14 < M < 2**14.
```

Hence the largest model number is 128.0 - 1.0/128. This model number is outside the range of the type and in consequence cannot be assigned to values of type F.

Ordinarily, the ACTUAL_DELTA value is choosen by the compiler, the only constraint being that it must be less than or equal to the delta for the type. In a respresentation specification, the user may specify the ACTUAL_DELTA value. By such a specification, the representation of values can be constrained to conform to external requirements. For instance, if an analogue to digital converter from a camera places values in the memory of the computer, it is important that the Ada program should use the same representation. Consider the case when values 0 to 127 are input in binary, but these are regarded as fractions of unit intensity from 0.0 to 127.0/128. Then one might have:

```
type INTENSITY is delta 1.0/128 range 0.0 .. 127.0/128;
for INTENSITY'ACTUAL_DELTA use INTENSITY'DELTA;
for INTENSITY'SIZE use 7;
```

Note that -1.0/128 is a model number of the type but that it cannot be stored in values because of the range constraint and in consequence, the sign is not needed in the representation.

As a further example of a representation specification, consider handling a type analogous to DURATION in Ada ie, timing intervals. The obvious representation is in clock ticks so that the ACTUAL_DELTA value might be 1.0/60 seconds. The programmer would wish to work in seconds to avoid changing the program to work in Europe (with 50 cycle mains supply). Hence one might have:

```
type DURATION is delta 1.0/60 range -24.0 .. 24.0;
for DURATION'ACTUAL_DELTA use DURATION'DELTA;
```

The special attributes of a fixed point types are as follows:

F'DELTA: A real literal value equal to the value of the expression after the 'delta'. In this case the value is 0.01. (30)

F'ACTUAL_DELTA: The real literal value used by the implementation as the constant for the multiples which give the model numbers. In this case the value is 1.0/128 = 0.0078125.

F'BITS: This is the number of bits needed to represent the unsigned model numbers. In this case, 7 bits is required before the point and 7 bits after making 14 in all, ie F'BITS=14. The value is that of the integer literals (see section 9).

F'LARGE: The largest model number of the type F. In this case the value is 128.0 - 1.0/128 = 127.992175. The value has the same type as that of real literals. In general, one has F'LARGE = (2**F'BITS - 1) * F'ACTUAL_DELTA.

Subtypes of fixed point types can be declared explicitly or implicitly by giving an accuracy constraint on the declaration of a variable. In an exactly analogous way to floating point, there is no run-time check for an accuracy constraint for fixed point. Consider:

```
subtype SF is F delta 0.02;
```

Note that the range constraint is not needed since the range of values is determined from the type definition (-100.0 .. 100.0 in this case). This subtype definition means that the set of model numbers is correspondingly reduced by the value SF'ACTUAL_DELTA being a binary power multiple of F'ACTUAL_DELTA. In this case, with F'ACTUAL_DELTA = 1.0/128, SF'ACTUAL_DELTA could be 1.0/64. The implementation need not reduce the model numbers for a subtype. For this reason, it is not permitted to set the ACTUAL_DELTA for a subtype in a representation specification. Hence the only action required by the compiler for the above subtype declaration is to check that the expression after delta has a value greater than or equal to F'DELTA. If an implementation does reduce the model numbers for a subtype, then the values SF'BITS and SF'LARGE reflect the value of SF'ACTUAL_DELTA.

Exercises

Given: type FX is delta D range L .. U;

(a) Are D, L and U model numbers? (31)

(b) Can the range constraint be omitted?

(c) If L < 1.0 < U, is 1.0 a model number?

What is wrong with the following?

(d) type FD is delta 0.01 range 0.0 .. SQRT(2.0);

(e) type FE is delta 0.01 range 0 .. 10;

(f) type FF is delta 10.0 range 0.0 .. 100*FF'DELTA;

8. The predefined fixed point operations

With floating point, the specification of each operation was easy since with the exception of **, the type of the operands and the result was the same. It is easy to see that in general this is not possible with fixed point. The rescaling of intermediate results is done by explicit type conversion. This rescaling is only essential on multiplication and division since the magnitude of values only changes significantly with these operations.

The operations which do not involve rescaling are tabulated below. Here X is of any fixed point type and I of any integer type, the result always being the same as X. (32)

Example	meaning
+ X	no operation
- X	change sign
X + X	addition
X - X	subtraction
I * X	equivalent to repeated addition
X * I	equivalent to repeated addition
ABS(X)	absolute value
X / I	division without rescaling

Now consider some examples of the calculation of error bounds for computation using the same example type F as above:

```
type F is delta 0.01 range -100.0 .. 100.0;
```

Given

```
ONE : F := 1.0;
```

then it is not safe to assume that ONE is represented exactly

unless it is known that F'ACTUAL_DELTA is a submultiple of 1.0. An implementation is free to truncate or round a constant on conversion to F, and in consequence all one can say is that ABS(ONE-1.0) <= F'DELTA. To simplfy the remaining discussion, assume that F'ACTUAL_DELTA is 1.0/128, Then, of course, 1.0 is a model number and therefore is stored exactly. Now consider

```
TENTH : F := 0.1;
```

Here the value is bounded by the model interval 12.0/128 .. 13.0/128, and could be either of the two extremes or a value in between. Now consider the expression 10*TENTH. This is equivalent to repeated addition and in consequence can yield any value in the interval 120.0/128 .. 130.0/128. Of course, on a binary machine, the expression will never yield 1.0 exactly since 0.1 has a recurring binary representation.

The fixed point operations follow the same logic as for floating point as far as the definition of error bounds is concerned. In consequence, all of the operations above except, possibly, division by an integer, will yield exact results for multiples of F'ACTUAL_DELTA assumming the result is in range. Since in some cases, the implementation will not have values between model numbers, this implies that all these operations are exact. In fact, values between model numbers can only arise from division by an integer, from constant, and type conversions from other types. The nature of the inexact operation can be illustrated from

```
X: F := 10.1;
Y: F := X/2;
```

The X value is bounded by the interval 1292.0/128 .. 1293.0/128. (33)
The Y value is then bounded by the interval 646.0/128 .. 647.0/128. Hence multiplication of Y by 2 will yield a larger bounding interval than that of X. If, of course, one knew that X was equal to an even multiple of F'ACTUAL_DELTA, then Y := X/2; and X := Y*2; would leave X unchanged. (*1) The model numbers for an integer type in fixed point operations are just the integers themselves.

The rescaling operations are general fixed point multiplication and division. The operands are of any, possibly distinct, fixed point types. Consider the types:

```
type F is delta 0.01 range -100.0 .. 100.0;
type G is delta 1.0 range -10_000.0 .. 10_000.0;
```

Given F1,F2:F, consider the product F1 * F2. This product is very likely to overflow the range of F and hence in general it cannot be considered to be of type F. Using the intuitive concept of formats, it is quite clear that a product has a different format.

(*1) If the statements Y:=X/2; X:=Y*2; appeared in a program, compiler optimization could give the exact result in all cases (since CONSTRAINT_ERROR cannot arise, optimization is safe).

On the other hand, given G1:G, it is clear that one should be able to assign the product to G1 without overflow. In Ada, the product is regarded as Universal Fixed - a hypothetical type of arbitrarily high accuracy. This type does not have an Ada name so that no variables can be declared of this type. All that can be done with such a product is to convert it into another type. In this case, one can write:

```
G1 := G(F1 * F2);
```

(34)

To calculate error bounds, the operation is regarded as a whole: ie calculating the product and the conversion. Assuming that F'ACTUAL_DELTA = 1.0/128 and G'ACTUAL_DELTA = 1.0, then one has:

F1	1.0	10.0	10.1
F2	2.0	0.1	10.1
bounds on F1	1.0..1.0	10.0..10.0	1292.0/128..1293.0/128
bounds on F2	2.0..2.0	12.0/128..13.0/128	1292.0/128..1293.0/128
bounds on F1*F2	2.0..2.0	120.0/128..130.0/128	101.88 .. 102.04
bounds on G1	2.0..2.0	0.0 .. 2.0	101.0 .. 103.0

In this case, the error bounds are about 2 units in the resulting type. Clearly, if the operands have high accuracy as well as the resulting type for the product, then no accuracy need be lost.

Fixed point division works in the same way with the requirement to convert to result of the operation. Naturally, if the right hand operand has a small value, then substantial inaccuracies can occur.

Exercises

(a) Given a fixed point type with a range of positive values only, can the function ABS have any use?

(b) Should an implementation limit the smallness of the delta value?

(c) What purpose does the 'DELTA value serve if errors are bounded by multiples of 'ACTUAL_DELTA?

(d) Given type FD is delta 0.01 range -1.0 .. 1.0; and assuming FD'ACTUAL_DELTA = 1.0/128, and X = 0.1, calculate the error bounds

on:

```
Y := 0.6 + FD(0.2*X) + FD(0.1*FD(X*X));
```

and on

```
Y := FD((FD(0.1*X) + 0.2)*X) + 0.6;
```

9. Literal expressions

In the preceeding sections one problem has been avoided, namely, the type of a numeric literal. It has been noted tnat literals are implicitly converted to the type required by the context. This implicit conversion can lose accuracy and can also raise the exception CONSTRAINT_ERROR if the value exceeds the implemented range of the type. (35)

Integer literals are regarded as being of type Universal Integer and real literals of type Universal Real. The names of these types are not available as Ada names and in consequence, cannot be used to declare variables etc. However, it often happens, especially with fixed point working, that there are relationships between literal values which cannot be easily expressed be means of typed expressions. Ada therfore allows for the evaluation (by the compiler) of literal expressions. Hence, wherever Ada permits an integer expression 1+1 (say), can be written. Each literal 1 is of type Universal Integer and the "+" is evaluated by the compiler independently of the context. Consider the following:

```
type INT is range -10 .. 10;
TEN: constant INT := 10;
THOUSAND: constant := 1000;
I : INT;

I := TEN;    -- OK
I := 1000 * TEN;    -- (1)
I := THOUSAND - THOUSAND;  -- (2) OK I := 0;
I := THOUSAND - 1000;   -- (3)
```

In case (1), the actions are as follows: 1000 is implicitly converted to INT, the INT "*" operation is applied, the result is checked for range, and lastly the assignment is performed. In this case, the first step fails as 1000 is not within the range of the type, and hence CONSTRAINT_ERROR is raised.

In cases (2) and (3), the result is to assign zero to I since the subtraction is that of Universal Integer which is performed by the compiler (within an unbounded range).

Similar remarks apply to real literal expressions:

```
type F is digits 5;
RATIO: constant F := 3.14;
```

```
PI: constant := 3.14159_26535;
F1 : F;

F1 := 2.0 * PI;  -- Universal Real multiplication

F1 := 2.0 * RATIO; -- Multiplication of type F

F1 := 1.0E200 - 10.0 * 1.0E199;
                -- Literal expression value 0.0
                --  no overflow possible
```

Of course, with both Universal Integer and Universal Real, a compiler will have some limitation in the size of values and accuracy respectively. These limits should not be of any practical significance and in consequence will be larger than any implemented type available on typical target machines.

The type Universal Real is not strictly a floating point or fixed point type but has the functionality of both. In consequence, all the following operations are legal as illustrated by the number declarations:

```
ADD: constant := 1.0 + 3.0;
SUB: constant := 6.0 - 8.0;
PLUS: constant := +12.0;
NEG: constant := - 8.4;
MULT1: constant := 2 * 3.0;
MULT2: constant := 4.0 * 10;
MULT3: constant := 5.0 * 10.1;
DIV1: constant := 10.0/2;
DIV2: constant := 10.0/2.0;
EXP: constant := 3.0 ** 2;
ABS1: constant := ABS(6.0);
```

(36)

The relational operators are also available for Universal Real giving the BOOLEAN result as usual. The semantics of these operations is the same as that for the typed operations except that the model intervals are smaller than any implemented real type. Intuitively, one can envisage a compiler storing values in a floating point type of high accuracy.

Exercises

Are the following literal expressions?

(a) F'DIGITS + 10
(b) FX'DELTA / 10.0
(c) FX'LAST - 0.1
(d) 1/FX'ACTUAL_DELTA

(37)

10. A floating point example, Blue's algorithm

Blue's algorithm [5] is for the calculation of the Euclidean norm of a vector (ie square root of the sum of the squares of the elements). It is a natural extension of the calculation of SQRT(A**2+B**2) which was used above. The algorithm is very carefully written to avoid overflow and underflow and also to guarantee the precision of the result. Hence it is a good example of a high-quality algorithm (orginally written in FORTRAN). The paper itself should be studied for the numerical analysis involved. (38)

The paper presents the algorithm in two forms: a mathematical formulation using Greek letters and conventional notation; and a formulation in RATFOR, a FORTRAN preprocessor. The major differences between the RATFOR version and that for Ada below are that the Ada version works for any implemented precision and does not depend on an additional subroutine to set critial constants. In Ada, these constants are model numbers and hence the definition of the language guarantees that the values are set correctly from the predefined attributes.

The identifiers used are those of the RATFOR implementation, but in upper case. In practice, longer identifiers that show the relationship to the mathematical formulation of the algorithm would be preferable.

The function itself, called NORM must assume an appropriate context for the types of its parameters and for the SQRT routine. This context is:

```
type REAL is digits D;
type VECTOR is array (INTEGER) of REAL;
function SQRT(X: REAL) return REAL;
```

The body of the function can now be given. The main logic is to go once through the vector accumulating three sums according to the magnitude of each element. The three sums are then scaled as appropriate to give the final answer.

```
function NORM(X: VECTOR) return REAL is

   -- calculate constants which depend upon REAL.
   -- Floor and ceiling functions of the paper avoided
   --    by using the truncation of integer division.
   EB1: constant := (REAL'EMAX + 1)/2; --  -(exponent of B1)      (39)
   B1 : constant REAL := 2.0 **(-EB1); -- model number of REAL

   EB2: constant := (REAL'EMAX - REAL'MANTISSA + 1)/2;
   B2 : constant REAL := 2.0 ** EB2;

   ES1M: constant := REAL'EMAX/2 + 1;
```

```
   S1M : constant REAL := 2.0 ** ES1M;

   ES2M: constant := (REAL'EMAX + REAL'MANTISSA + 1)/2;
   S2M : constant REAL := 2.0 ** (-ES2M);
   OVERFL: constant REAL := REAL'LARGE * S2M;
               -- this value can be calculated by the compiler

   RELERR: constant REAL := SQRT(REAL(REAL'EPSILON));
           -- Conversion necessary as 'EPSILON is a literal.
           -- Note that this value must be calculated dynamically.

   ABIG, AMED, ASML: REAL := 0.0; -- the three accumulators
   AX: REAL;
   N: constant INTEGER := X'LENGTH;
begin
-- size of array = 0 is not special case in Ada (unlike FORTRAN)
if N > 2**REAL'MANTISSA then                                        (40)
   raise CONSTRAINT_ERROR;  -- not clear how to handle this case
end if;
for J in X'FIRST .. X'LAST loop
   AX := ABS(X(J));
   if AX > B2 then
      ABIG := ABIG + (AX * S2M)**2;
   elsif AX < B1 then
      ASML := ASML + (AX * S1M)**2;
   else
      AMED := AMED + AX**2;
   end if;
end loop;
if ABIG > 0.0 then
   ABIG := SQRT(ABIG);
   if ABIG > OVERFL then
      return REAL'LARGE; -- can't raise NUMERIC_ERROR as well as
                         -- returning a result
   end if;
   if AMED > 0.0 then
      ABIG := ABIG / S2M;
      AMED := SQRT(AMED);
   else
      return ABIG/S2M;
   end if;
elsif ASML > 0.0 then
   if AMED > 0.0 then
      ABIG := SQRT(AMED);
      AMED := SQRT(ASML)/S1M;
   else
      return SQRT(ASML)/S1M;
   end if;
else
   return SQRT(AMED);    -- the standard path
end if;
if ABIG > AMED then
   ASML := AMED;
```

```
else
   ASML := ABIG;
   ABIG := AMED;
end if;
if ASML <= ABIG * RELERR then
   return ABIG;
else
   return ABIG * SQRT( 1.0 + (ASML/ABIG)**2 );
end if;
end NORM; --  (*1)
```

(41)

The algorithm is not completely satisfactory in the sense that although it will work for any real type, the algorithm uses only the Ada properties of the type. By replacing each occurance of 'REAL by REAL'BASE, the algorithm would use the properties of the implemented type. With such a replacement, there would only be one effective version of the function for each of the hardware types. Further 'improvements' can be made by exploiting specific machine-dependent properties of the type (see section 13).

(42)

11. A fixed point example

A common requirement in fixed point is to mimic floating point to conserve either time or space. The evaluation of a polynomial is sometimes used to approximate a function. Such polynomials are typically truncated power series which rely upon the decreasing contributions from the higher order terms. Given:

(43)

```
Y := A + B*X + C*X**2 + D*X**3;
```

the most effective evaluation method is nested multiplication, ie

```
Y := ((D*X + C)*X + B)*X + A;
```

As X is small, with floating point, the normalization on the addition of A is the only source of rounding error. With fixed point using a pure fraction as the data type, each partial product can be calculated with minimal errors so that the resulting error is again minimised. Note that performing the calculation in polynomial fashion both involves more operations and is, in general, less accurate.

To illustrate the use of fixed point for approximation, the calculation of the sine and cosine functions is given from Cody and Waite [6]. It is assumed that floating point is expensive on the target machine and therefore the major computation uses fixed point. The algorithm illustrates a number of other features including type conversion, literal expressions, integer type definitions and conditional compilation. The algorithms given in [6] include a number of options which would ordinarily be chosen by the implementor. Some of the choices in this case are inserted into the algorithm so that the compiler selects the necessary code.

(*1) Not yet tested with an Ada compiler

In order to correspond to conventional usage, the routines SIN and COS are coded for the type FLOAT. No assumptions are made about the type except that integer and fixed point types are available with sizes about the same as the mantissa length of the type FLOAT.

The argument reduction is a difficult aspect of sine and cosine. The constants C1 and C2 (whose sum is PI) are used for this in the way recommended for a machine without guard digits on floating point. The code illustrates that compiler "optimization" of floating point can be very unsafe.

```
package SIN_COS is
    function SIN( X: FLOAT) return FLOAT;
    function COS( X: FLOAT) return FLOAT;
end SIN_COS;

package body SIN_COS is
    PI: constant := 3.14159_26535_89793_23846;                    (46)
    PI_DIV_2: constant := PI/2;
    ONE_DIV_PI: constant := 1.0/PI;
    SGN_POS: BOOLEAN;
    Y: FLOAT;

procedure COMMON_PART( X: FLOAT );

function SIN( X: FLOAT) return FLOAT is
    begin
    if X < 0.0 then
        SGN_POS := FALSE;
        Y := - X;
    else
        SGN_POS := TRUE;
        Y := X;
    end if;
    COMMON_PART(X);
    return Y;
    end SIN;

function COS( X: FLOAT ) return FLOAT is
    begin
    SGN_POS := TRUE;
    Y := ABS(X) + PI_DIV_2;
    COMMON_PART(X);
    return Y;
    end COS;

procedure COMMON_PART( X: FLOAT ) is
    B: constant := FLOAT'MANTISSA;                                (46)

    type INT is range 0 .. 4 * 2**(B/2);
    YMAX: constant INT := INT(PI*2**(B/2)+0.5);
```

```
N: INT;

X1, X2, XN, F: FLOAT;
C1: constant FLOAT := 8#3.1104#;
C2: constant FLOAT := -8.9089_10206_76153_73566_17E-6;
EPS: constant := 2.0 ** (B/2);

D: constant := 2.0 ** (-B);
type FR is delta D range -1.0 + D .. 1.0 - D;
G: FR;

begin
if Y >= FLOAT(YMAX) then
    raise CONSTRAINT_ERROR;
else
    N := INT( Y * ONE_DIV_PI );
    XN := FLOAT(N);
    if N mod 2 = 1 then
        SGN_POS := not SGN_POS;
    end if;
    if ABS(X) /= Y then
        XN := XN - 0.5;  -- COS wanted
    end if;
    X1 := FLOAT(INT(ABS(X)));
    X2 := ABS(X) - X1;
    F := ((X1 - XN*C1) + X2) - XN*C2;                    (47)
    if ABS(F) < EPS then
        Y := F;
    else
        G := FR(F/2.0);
        G := FR(G * G);
        if B <= 24 then                                  (48)
            Y := FLOAT(
                    FR((
                      FR((
                        FR((
                          FR(0.00066_60872 * G)
                        - 0.01267_67480) * G)            (44)
                      + 0.13332_84022) * G)
                    - 0.66666_62674) * G)
                    );
        elsif B <= 32 then
            Y := FLOAT(
                    FR((
                      FR((
                        FR((
                          FR((
                            FR(-0.00002_44411_867 * G)
                          + 0.00070_46136_593) * G)
                        - 0.01269_81330_068) * G)
                      + 0.13333_32915_289) * G)
                    - 0.66666_66643_530) * G)
                    );
```

```
            elsif B <= 50 then
                Y := FLOAT(
                        FR((
                          FR((
                            FR((
                              FR((
                                FR((
                                  FR((
                                    FR(-0.00120_76093_891E-5 * G)
                                  + 0.06573_19716_142E-5) * G)
                                - 0.00002_56531_15784_674) * G)
                              + 0.00070_54673_00385_092) * G)
                            - 0.01269_84126_86862_404) * G)
                          + 0.13333_33333_32414_742) * G)
                        - 0.66666_66666_66638_613) * G)
                        );
            elsif B <= 60 then
                Y := FLOAT(
                        FR((
                          FR((
                            FR((
                              FR((
                                FR((
                                  FR((
                                    FR(0.00001_78289_31802E-5 * G)
                                  - 0.00125_22156_53481E-5) * G)
                                + 0.06577_74038_64562E-5) * G)
                              - 0.00002_56533_57361_43317) * G)
                            + 0.00070_54673_71779_91056) * G)
                          - 0.01269_84126_98369_17789) * G)
                        + 0.13333_33333_33330_64050) * G)
                      - 0.66666_66666_66666_60209) * G)
                      );
             else
                 raise CONSTRAINT_ERROR;
             end if;
             Y := F + F*R;
         end if;
         if not SGN_POS then
             Y := - Y;
         end if;

     end COMMON_PART;

end SIN_COS;  --  (*1)
```

(*1) Not yet tested with an Ada compiler

12. The complex data type - an example of generics

An essential difficulty with both fixed point and floating point subroutines is that they can only be used with one type - and in consequence one accuracy. Usually, the text of a subroutine will be more general than this although when it is compiled, it will be for a specific accuracy. The full generality of the source text can be exploited by means of generics. The numeric types are made generic parameters so that specific instantiations give any specific accuracy (supported by the implementation).

As an example of generics, the package for providing complex data types is used. This is very similar to the rational number package given in the language reference manual (which does not use generics).

```
generic
    type REAL is digits <>;  -- matches any floating point type
package COMPLEX_OPS is
    type COMPLEX is record
                    RE, IM: REAL;
                    end record;
    function "-" ( X: COMPLEX) return COMPLEX;
    function ABS( X: COMPLEX) return REAL;
    function "+" ( X, Y: COMPLEX ) return COMPLEX;
    function "-" ( X, Y: COMPLEX ) return COMPLEX;
    function "*" ( X, Y: COMPLEX ) return COMPLEX;
    function "/" ( X, Y: COMPLEX ) return COMPLEX;
end COMPLEX_OPS;
```

The package body does not repeat the generic parameters, and could be:

```
with MATH_LIB;
package body COMPLEX_OPS is

    function "-" ( X: COMPLEX) return COMPLEX is
        begin
        return ( - X.RE, - X.IM );
        end "-";

    function ABS( X: COMPLEX) return REAL is
        A, B: REAL;
        begin
        if ABS(X.RE) > ABS(X.IM) then
            A := ABS(X.RE);
            B := ABS(X.IM);
        else
            A := ABS(X.IM);
            B := ABS(X.RE);
        end if;
        if A > 0.0 then
```

```
            return A * MATH_LIB.SQRT(1.0 + (B/A)**2);
        else
            return 0.0;
        end if;
        end ABS;
    function "+" ( X, Y: COMPLEX ) return COMPLEX is
        begin
        return ( X.RE + Y.RE, X.IM + Y.IM );
        end "+";

    function "-" ( X, Y: COMPLEX ) return COMPLEX is
        begin
        return ( X.RE - Y.RE, X.IM - Y.IM );
        end "-";

    function "*" ( X, Y: COMPLEX ) return COMPLEX is
        begin
        return ( X.RE*Y.RE - X.IM*Y.IM,
                 X.IM*Y.RE + X.RE*Y.IM );
        end "*";

    function "/" ( X, Y: COMPLEX ) return COMPLEX is
        A: REAL := Y.RE**2 + Y.IM**2;
        begin
        return ( (X.RE*Y.RE + X.IM*Y.IM) / A,
                 (X.IM*Y.RE - X.RE*Y.IM) / A );
        end "/";

end COMPLEX_OPS;  --  (*1)
```

(50)

13. Portability Issues

The Ada language does not aim at complete portability. To do so would mean that it would be impossible to write machine-specific code such as that illustrated in section 11. Also, the differences in actual hardware does not make portability achievable at acceptable costs. Ultimately, the floating point addition of a machine is defined by the microcode, which cannot even be characterized by a few simple parameters. Hence programmers need to be aware of potential portability problems so that code is not needlessly machine-specific.

(51)

Integer Types.
New types should be introduced for reasons of abstraction and modularization. It is particularly important to introduce new integer types in handling large ranges (> 16 bits) since these may not be supported, or will have some significant penalty. If a large integer type is only used in one small routine, then recoding is much simpler than if INTEGER is used thoughout (and

---

(*1) Not yet tested with an Ada compiler

only during execution it is found that one routine assumes INTEGER'LAST > 2**16).

A trap for the unwary is that intermediate results in an expression of type T can exceed the range T'FIRST..T'LAST. In consequence, portability is not assured since on other hardware T may correspond exactly to the range of a hardware type. Diagnostic compilers can trap this condition and cause NUMERIC_ERROR to be raised at run-time. Of course, the values for which NUMERIC_ERROR is raised is again dependent upon the hardware. On some machines, calculations are done to 32 bits but stored values are 16 bits giving the overlength accumulator problems analogous to floating point.

Floating Point Types.
Similar difficulties arise with the handling of NUMERIC_ERROR for floating point. The actual range of the implemented type is likely to exceed the range -F'LARGE .. F'LARGE due to having a larger exponent than that guaranteed by Ada. A machine may have 'infinite' values such as those of the IEEE standard [7], in which case substantial care is necessary to ensure such facilities are avoided or used in a portable fashion. For this reason, the values F'FIRST and F'LAST should be avoided. It must be remembered that the NUMERIC_ERROR exception might never be raised.

Explicit use of the type FLOAT should be restricted to small sections of code. It would be reasonable to assume that FLOAT has 5 digits of precision. Of course, some machines may not have any floating point. Apart from the use as a tool for abstraction, new types should be introduced when different accuracies are needed. The use of accuracy constraints in subtypes should only be regarded as a comment, rather than attempting to rely upon sophisticated optimization.

A set of machine specific attributes for floating point is available which, if used, is unlikely to give portable code. For this reason, these attributes have names beginning with MACHINE_. They are as follows:

F'MACHINE_RADIX. This is the radix used to represent machine values. To support the Ada model of floating point properly, it must be a power of 2. It is 16 for the IBM 360/370 and 2 for the IEEE standard.

F'MACHINE_MANTISSA. This is the mantissa length in radix units. It is at least conceivable that there is not a whole number of radix places in the mantissa, although the value is defined to be an integer.

F'MACHINE_EMAX. The maximum exponent value in radix units. (So F'MACHINE_RADIX ** F'MACHINE_EMAX is

approximately the largest machine number.)

F'MACHINE_EMIN. The minimum exponent value in radix units.

F'MACHINE_ROUNDS. A BOOLEAN value which is true only if all floating point operations perform true rounding, such as tnat of the IEEE standard.

F'MACHINE_OVERFLOWS. A BOOLEAN value which is true only if the NUMERIC_ERROR exception is raised whenever the result of an operation cannot be represented with the usual precision due to exceeding the range of machine values.

With care, the MACHINE_RADIX value can be used to overcome the problem of 'wobbling precision' and the attribute MACHINE_OVERFLOWS can be used to provide an alternative coding which relies upon the NUMERIC_ERROR exception. Note that one cannot easily determine the largest and smallest positive machine values due to the differences between 1's and 2's complement arithmetic, underflow etc.

Fixed Point Types.
A similar remark applies to fixed point about not relying upon values outside the range -LARGE..LARGE. A program can also depend upon the arithmetic of the machine. With a pure fraction, on a 2's complement machine, -1.0 is a machine number but 1.0 is not, whereas neither are machine values on a 1's complement machine. Although it is highly likely that the default value for ACTUAL_DELTA will be a power of two, the program should not rely upon this.

In the same way that one should not rely upon the existance of floating point with a large number of digits, so with fixed point one should not expect very high precision (exceeding 32 bits, say).

Fixed point types have the attribute FX'MACHINE_ROUNDS which is true only if all the operations perform true rounding.

References

[1] W S Brown. "A realistic model of floating point computation" Mathematical Software III ed. J Rice pp 343-360 Academic Press New York 1977.

[2] W S Brown. "A simple but realistic model of floating point computation". Computer Science Technical Report No. 83. May 1980. Bell Laboratories, Murray Hill, NJ 07974.

171

[3] W S Brown and S I Feldman. "Environmental parameters and basic functions for floating point computation". Computer Science Technical Report No. 12, March 1980, Bell Laboratories, Murray Hill, NJ 07974

[4] J D Ichbiah et al. "Reference Manual for the Ada programming language". Department of Defense, Washington, July 1980.

[5] J L Blue. "A portable FORTRAN program to find the Euclidean norm of a vector." ACM TOMS Vol4, No1 pp15-23 March 1978.

[6] W J Cody and W Waite. "Software manual for the elementary functions." Prentice-Hall. Englewood Cliffs, NJ 07632. 1980.

[7] "A Proposed Standard for Binary Floating Point Arithmetic" SIGNUM Newsletter, October 1979. (Now to be adopted as an IEEE Standard.)

Answers to exercises

Page 5 (section 2)

16#FF# = 15 * 16 + 15 = 240 + 15 = 255
4#1.01#E2 = 4#101.0# = 4.0 ** 2 + 1.0 = 17.0
3#0.1# = 3.0 ** (-1) = 0.33333 (no exact decimal equivalent)
8#0.1# = 8.0 ** (-1) = 0.125
16#0.8# = 8* 16.0 ** (-1) = 0.5
16#0.999999...# = 9.0/(16-1) = 3.0/5 = 0.6
3._14 -- no underscore after decimal point
4#_0.1#2 -- no underscore after sharp
16#FF#E-1 -- not integer valued and no decimal point
8#0.9# -- 9 not a radix character.

Page 7 (section 3)

(a) Next model number above 1.0 is 16#0.8001# * 2
=16#1.0002#

(b) Next model number below 1.0 is 16#0.7FFF8#
= 16#0.FFFF#

(c) The ratio is 2.0 which is the radix.

The rational numbers which cannot be represented exactly are those with recurring binary representation.

Page 13 (section 4)

Not necessarily. If F'DIGITS = 14 then F'MANTISSA = 47 but G'DIGITS = 7 gives G'MANTISSA = 24.

X = F'LARGE since 1.0/F'LARGE > F'SMALL and hence does not underflow. Of course, the actual machine may permit larger and smaller values without over/underflow.

Page 16 (section 5)

(A+B)+C=A+(B+C) is not necessarily true since floating point addition is not associative. For values A, B and C which are model numbers such that the true sum (and partial sums) are model numbers, the result will be true.

A+B=B+A is usually true but is not necessarily so. A+B could be calculated in the accumulator of the machine and then stored while B+A is evaluated in the accumulator. The comparison may then fail if the accumulator gives more precision than that of stored values.

```
A + 12 -- 12 is not real, must write A + 12.0
24 * B -- 24 is not real, must write 24.0 * B
C ** 2.0 -- exponent must be integer, hence should be C**2
```

Page 18 (section 6)

F'BASE'DIGITS >= F'DIGITS
F'BASE'LARGE >= F'LARGE

Not necessarily, since if the short form is used (without new), the compiler is free to choose the hardware type which need not be the same.

Page 21 (section 7)

(a) Not necessarily, but there must be model numbers close to L and U.
(b) No, it is required to determine the representation.
(c) Not necessarily, if the actual_delta is not a submultiple of 1.0, then 1.0 will not be a model number. The actual_delta value could exceed 1.0.
(d) The range must be static, hence the call of SQRT is not permitted.
(e) The range is a real range and hence should read "0.0 .. 10.0".
(f) The attribute 'DELTA is not defined until the end of the type definition and hence cannot be used within the definition.

Page 22 (section 8)

(a) Yes, values of the type always include negative values since model numbers can be negative. These negative values may cause CONSTRAINT_ERROR on assignment. If the size of a small value is in doubt, ABS can be used before assignment.

(b) There should be no practical lower limit. An implementation is likely to limit the number of model numbers for a type so that values can be held in one or two words. Hence if the delta value is very small, the L and U values should be also.
(c) Since 'DELTA >= 'ACTUAL_DELTA, the bounds can be expressed in terms of DELTA (ie the type definition).
(d) In both cases, 0.6 is of type FD and bounded by 76.0/128 .. 77.0/128. The constants 0.1 and 0.2 are Universal Real and in the first case are held to the relative accuracy of FD ie, 7 bits. This implies that 0.1 is bounded by an interval of width 1.0/(8*128) and 0.2 by an interval twice that width.

```
CASE 1
0.6      bounded by       76.0/128 .. 77.0/128
FD(0.2*X)   bounded by      2.0/128 .. 3.0/128
P=FD(X*X)   bounded by      1.0/128 .. 2.0/128
FD(0.1*P)   bounded by      0.0/128 .. 1.0/128
Y           bounded by   (76.0+2.0+0.0)/128 .. (77.0+3.0+1.0)/128
                    =     78.0/128 .. 81.0/128

CASE 2
Q=FD(0.1*X)    bounded by 1.0/128 .. 2.0/128
R=Q + 0.2      bounded by 26.0/128 .. 28.0/128
T=FD(R*X)      bounded by 2.0/128 .. 3.0/128
Y=T+0.6        bounded by 78.0/128 .. 80.0/128
```

The effectiveness of nested multiplication increases with the number of terms, as can be seen from the relative error of the higher order terms.

Page 25 (section 9)

(a) Yes, F'DIGITS is Universal Integer.
(b) Yes, FX'DELTA is Universal Real.
(c) No, FX'LAST is of type FX.
(d) No, Universal_Integer/Universal_Real is not permitted.

Real Data Types in Ada

B. A. Wichmann.

National Physical Laboratory

UK

Real = Fixed or Floating Point

	FORMATS	
Fixed Point	± dd.d	± ddd.d
Floating Point	± d.dd E ± dd	± d.ddd E ± dd

Data Type defines FORMAT

Fixed point ± d.dd

Examples: half +0.50 stored exactly

third +0.33 π +3.14 error < 0.01

Floating point ± d.dd E ±d

Examples: hundred +1.00E+2 stored exactly

π +3.14E+0 third +3.33E-1 error < 1 part in 1000

177

Fixed point - absolute error bound

Floating point - relative error bound

Some values stored exactly

Data Types to distinguish logically separate data.

Hence even if FEET and DEGREES both require format ±ddd.d, distinct types should be used.

Decimals formats are used for illustration only.
(Ada uses binary).

Notation for numeric literals in Ada

Integers - sequence of digits with underscore
(plus optional exponent)

Real literals - with decimal point

Six million

integer	6_000_000	6E6	6_000E+3
real	6_000_000.0	6.0E6	6_000.0E+3

Bases other than ten

2 # 101 # = $2^2 + 2^0 = 5$ integer

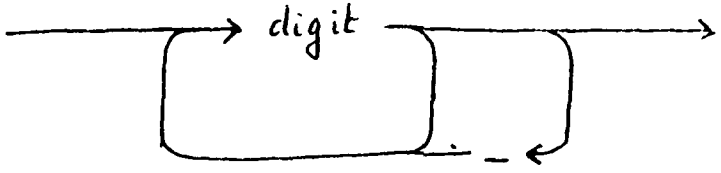
↑ base ↓ base digits

4 # 1.1 # = $4.0^1 + 4.0^{-1} = 1.25$ real

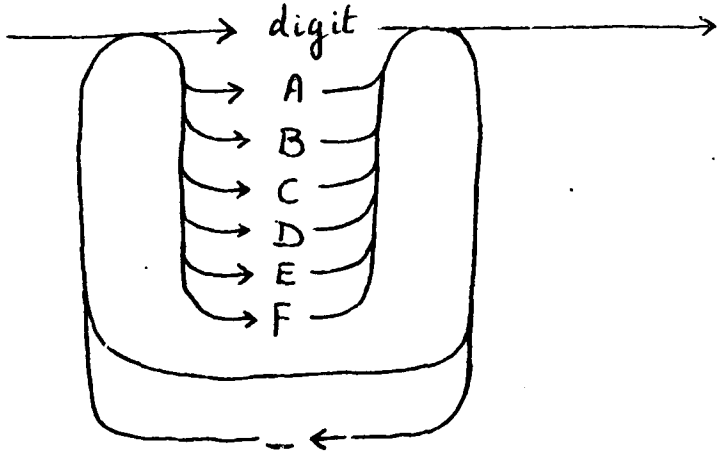
2 # 1.0 # E-20 = $2.0^{-20} \simeq 9.54E\text{-}7$ real

16 # F F # = $15 * 16^1 + 15 * 16^0 = 255$ integer

integer:

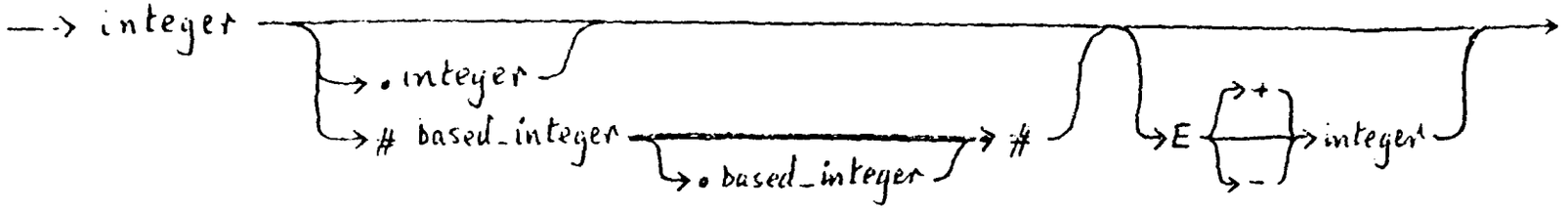


based_integer:



181

numeric_literal



number_declaration

→ identifier : constant := numeric_literal ; →

Example

```
PI : constant := 3.14159_26535_89793;
```

Exercises

What values are these in decimal?

16#FF# 4#1.01#E2 3#0.1# 3#0.1# 16#0.8#

What value is 16#0.999999# just a bit less than?

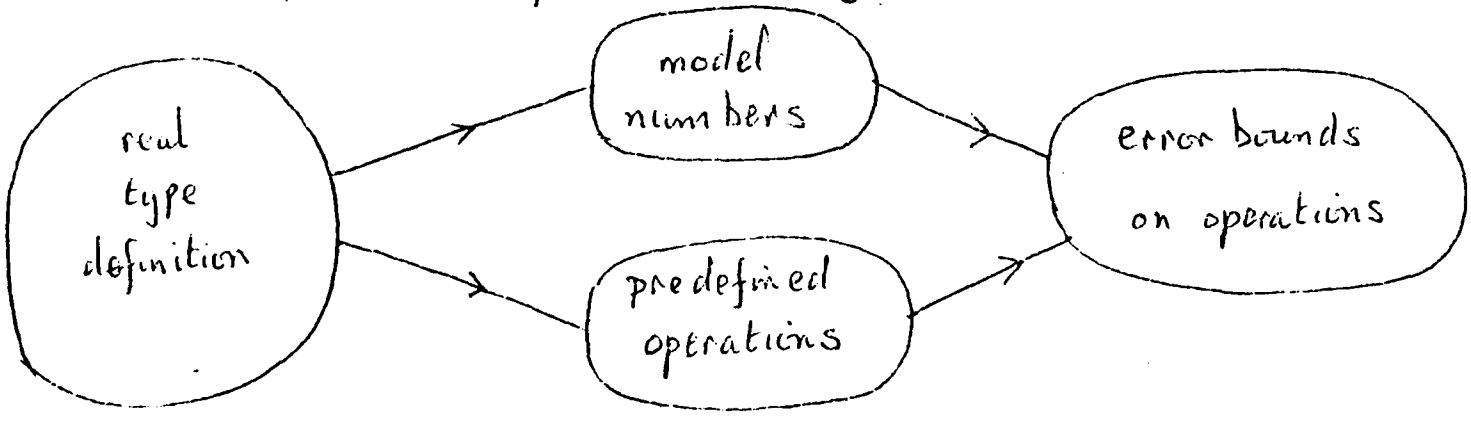
What is wrong with the following?

3._14 4#_0.1#2 16#FF#E-1 8#0.9#

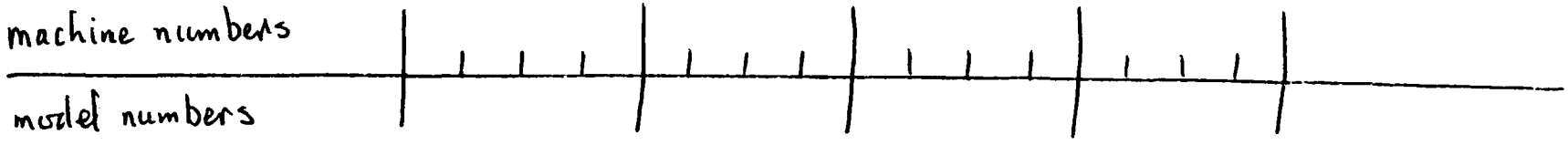
(Answers at back of notes)

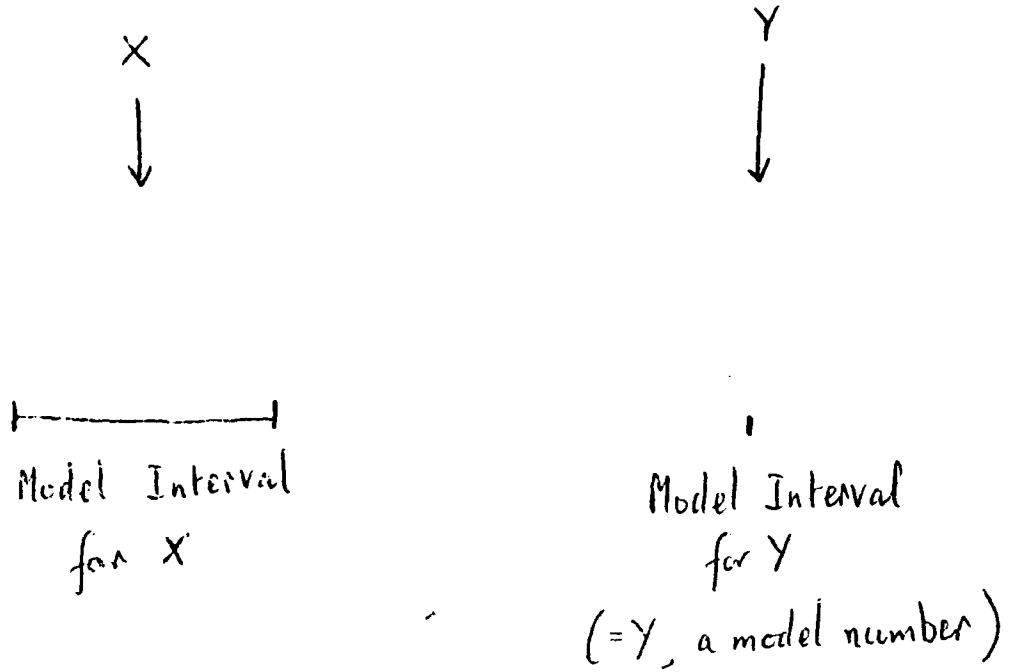
Ada properties of real types

- Permissive - different implementations allowed.
- Subset of values held exactly - model numbers
- Intervals bounded by model numbers bound errors in operations
- Model numbers defined in binary



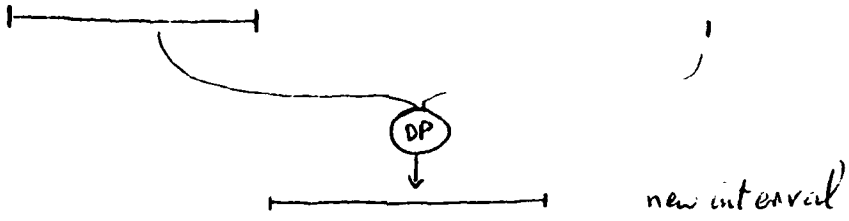
Accuracy of real operations





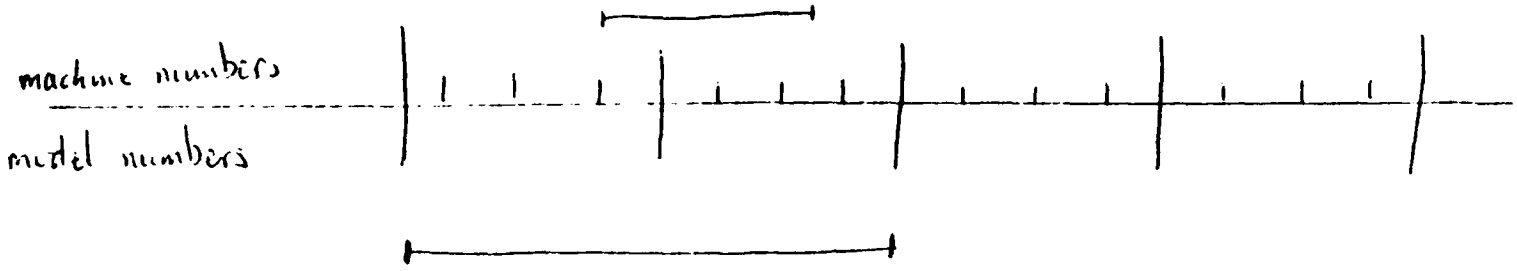
STEP 1, widen operands to model intervals

(to be placed on top of 11)



STEP 2, Perform operation mathematically to the intervals

(to be placed on top of II, without IIA)



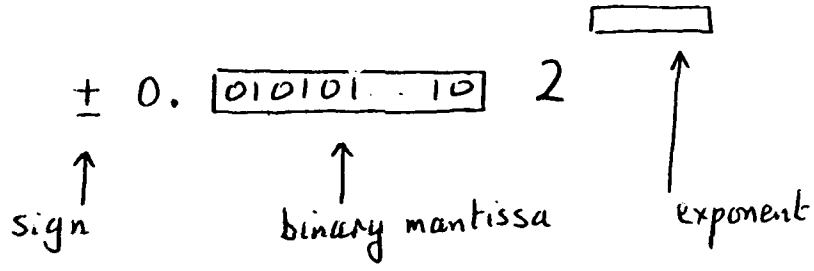
STEP 3 Widen resulting interval to a model interval.

This interval bounds the implemented operation.

Corollary 1: Floating point operations on small integers giving small integer results are exact.

Corollary 2: Classical error analysis can be applied.

Model Numbers for Floating Point



Type definition gives decimal digits of accuracy D.

implies D * log (10)/log (2) binary places

= B

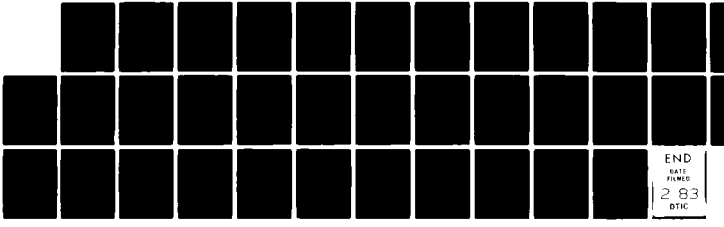
Exponent range taken to be (at least)

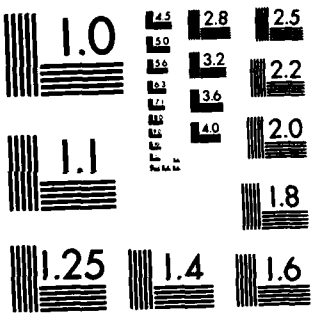
- 4 * B .. 4 * B

(compromise based upon current hardware).

AD-A124 012 USING SELECTED FEATURES OF ADA: A COLLECTION OF PAPERS (U) BATTELLE COLUMBUS LABS OH N HABERMAN ET AL. 09 NOV 82 DAAG29-76-D-0100 3/3

UNCLASSIFIED F/G 9/2 NL





Example

type F is digits 5;

F'DIGITS = 5

requires 5 * 3.32 = 16.6 binary places

Hence B = 17

F'MANTISSA = 17

Exponent range is -68 .. 68

F'EMAX = 68

Smallest model number > 0.0 is

2.0 ** (-69) = F'SMALL

Largest model number is

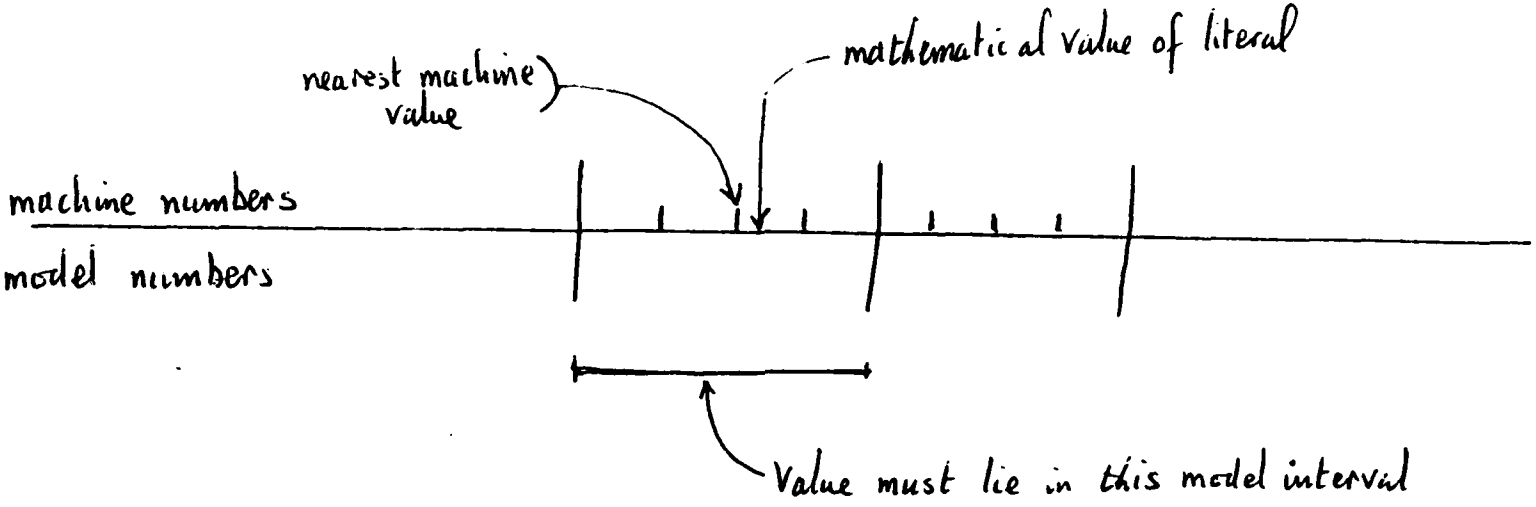
2#0.11111_11111_11111_11#E68

= 2.0 **68 - 2.0 **51 = F'LARGE

Gap between 1.0 and next model number above 1.0

= 2.0 ** (-16) = F'EPSILON

Conversion of literals



Corollary: Literals whose value is a model number are stored exactly.

Under flow $|X| < F'SMALL$

Model intervals are $-F'SMALL .. 0.0$

and $0.0 .. F'SMALL$

Value produced could be 0.0 or $\pm F'SMALL$

(large relative error)

Example:

```
X := SQRT ( A **2 + B **2);
```

Small values for A and B will give X = 0.0 (or F'SMALL?)

Can be overcome by special coding if necessary.

(See Notes)

OVERFLOW

Occurs in steps 1 and 3 in the definition of the bounding model interval. Values X, such that |x| > F'LARGE have no bounding interval.

Several things can happen:

(a) Due to large exponent range, errors are bounded properly on actual hardware

(b) NUMERIC_ERROR exception is raised

(c) "infinite" values are produced by hardware. (These are not in any model interval)

Conversion of literal values will raise CONSTRAINT_ERROR if out of range of implemented type.

NUMERIC_ERROR and CONSTRAINT_ERROR

Exceptions are propogated through procedure calls

Hence specification of procedure should state

(a) Exceptions cannot be raised

(b) Local handlers ensure exceptions not propogated

(c) Exceptions can be raised (circumstances?)

Subtypes of floating point types
(named subtypes or on individual objects)

- Range constraint with dynamic values for the bounds
  (run-time checking)
- Accuracy constraint digits N (<= F'DIGITS)

  Mantissa length of model numbers reduced
  Exponent range the same
  Hence
  SF'MANTISSA <= F'MANTISSA
  SF'SMALL = F'SMALL
  SF'EMAX = F'EMAX etc
  SF'EPSILON >= F'EPSILON
- Implementations are unlikely to make use of restricted accuracies for subtypes

Exercises

Given two floating point types such that

F'DIGITS = 2 * G'DIGITS

does F'MANTISSA = 2 * G'MANTISSA ?

What is the largest X such that X does not overflow nor does 1.0/X underflow ?

(Answers at back of notes)

Predefined Floating Point Operations

X, Y : FLOAT

+ X	no operation
- X	change sign
X + Y	addition
X - Y	subtraction
X * Y	multiplication
X / Y	division
ABS(X)	absolute value

(I : INTEGER

X ** I	exponentiate)

All these operations follow standard rules for bounding errors

Example 1 X, Y : F ; F'DIGITS = 5

X / Y with X = 15.0 and Y = 3.0

Both values are model numbers as is the mathematical result, in consequence, the machine result is exactly 3.0

Example 2

X * Y with X = 0. and Y = 10.0

X in model interval 16#0.19999# .. 16#0.1999A#

Y is a model number

Step 2 gives 16#0.FFFFA# .. 16#1.00004#

Step 3 gives 16#0.FFFF8# .. 16#0.10001#E1

Example 3

X + Y X = 1.0 Y = F'SMALL

Result in model interval 1.0 .. 1.0 + F'EPSILON

(as F'EPSILON > F'SMALL)

Exponentiate operator

(any floating point expression) ** (any integer expression)

Defined by repeated multiplication

X **2 equivalent to X *X

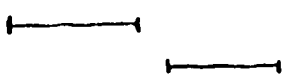
X **(-2) equivalent to 1.0/(X *X)

X **0 equivalent to 1.0

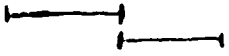
551

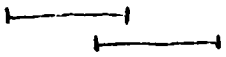
Relational Operators

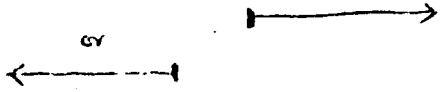
Result depends upon relationship between model intervals of each operand

(a)  correct result

(b) correct result

(c)  correct result or X op X

(d)  implementation defined

(e) 

implementation defined
or NUMERIC_ERROR

Exercises

A, B, C : FLOAT;

Does $(A+B)+C = A+(B+C)$?

$A+B = B+A$?

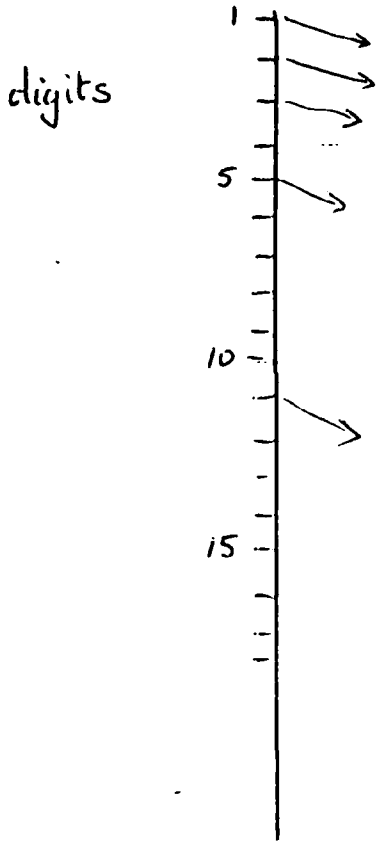
What is wrong with the following?

A + 12

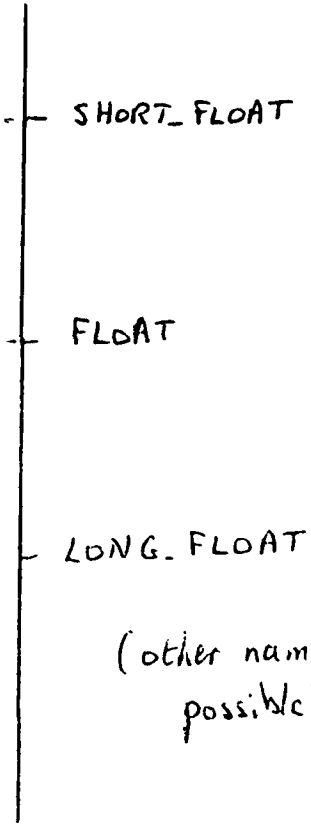
24 * B

C ** 2.0

Ada floating point types



Hardware Types



(other names are possible)

Mapping used will depend upon implementation

type F is digits D;

shorthand for

type F is new FLOAT digits D;

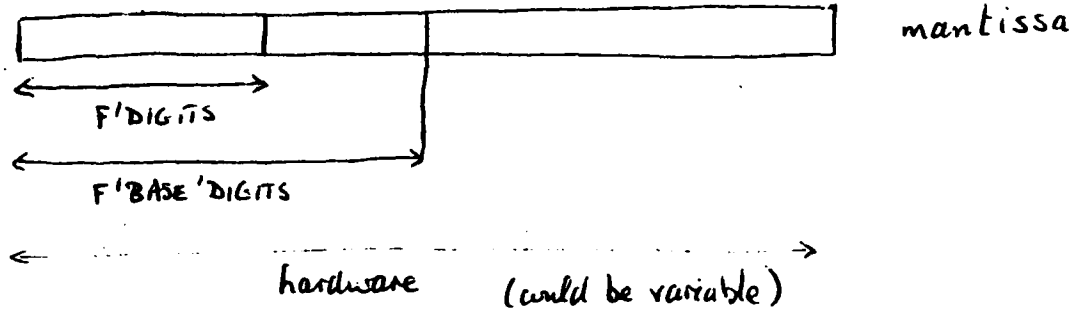
↑ appropriate hardware type

F is said to be *derived* from FLOAT.

Hardware type is 'BASE, which can be used in attributes

i.e. FLOAT'DIGITS = F'BASE'DIGITS

of course F'BASE'DIGITS >= F'DIGITS



Fixed Point Data Types

type Fx is delta D range L .. U;

- D — FX'DELTA
- L — FX'FIRST
- U — FX'LAST

Model numbers are integer multiples of FX'ACTUAL_DELTA.

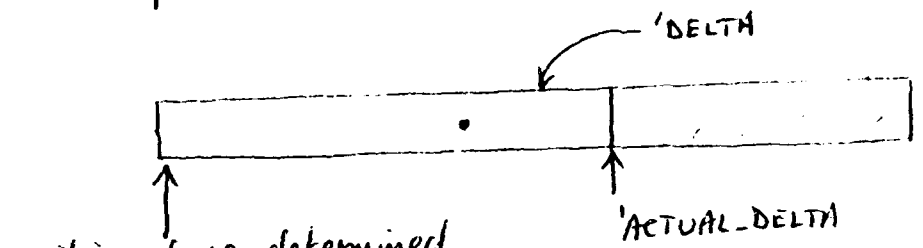
FX'ACTUAL_DELTA <= FX'DELTA

Sign * multiple * FX'ACTUAL_DELTA

0 <= multiple <= $2^N - 1$ (for some N)

A representation specification allows the programmer to specify the ACTUAL_DELTA.

Representation used for fixed point



Unlike floating point, a representation specification can remove the extra accuracy of the hardware.

```
type INTENSITY is delta 1.0/128 range 0.0 .. 127.0/128;
for INTENSITY'ACTUAL_DELTA use INTENSITY'DELTA;
for INTENSITY'SIZE use 7;
```

Not all model numbers can be represented as stored values because of a range constraint

Attributes of Fixed Point Types

F'DELTA - real literal value in type/subtype declaration

F'ACTUAL_DELTA - real literal value of the type/subtype F

F'BITS - integer value, the number of bits in an unsigned model number

F'LARGE - largest model number of type F

= (2 ** F'BITS - 1) * F'ACTUAL_DELTA

Accuracy constraints can be applied to fixed point types to give subtypes.

Implementation can, but need not, have different representation for subtypes

Exercises

type FX is delta D range L .. U;

(a) Are D, L, U model numbers?

(b) Can the range constraint be omitted?

(c) If L < 1.0 < U, is 1.0 a model number?

What is wrong with:

(d) type FD is delta 0.01 range 0.0 .. SQRT(20);

(e) type FE is delta 0.01 range 0 .. 10;

(f) type FF is delta 10.0 range 0.0 .. 100 * FF'DELTA;

Fixed Point operations without rescaling

X, Y : FX,

result type FX

I any integer type

+ X	no operation
- X	change sign
X + Y	addition
X - Y	subtraction
I * X	equivalent to repeated addition
X * I	equivalent to repeated addition
ABS(X)	absolute value
X / I	division without rescaling

Literal values converted to within 'ACTUAL_DELTA of true value

Examples of error bounds on computations

(F'ACTUAL_DELTA = 1.0/128)

```
X : F := 10.1 ; --bounded by 1292.0/128 .. 1293.0/128
Y : F := X/2 ; -- bounded by 646.0/128 .. 647.0/128
Z : F := 2 * X; -- bounded by 2584.0/128 .. 2586.0/128
W : F := 10.1 + X ;  -- same bounds as Z
```

Conversion of literal values outside the range of the implemented type will raise CONSTRAINT_ERROR.

Operations giving values outside the range of the implemented type will raise NUMERIC_ERROR.

Fixed Point rescaling operations

Scale needed for result of multiplication and division cannot be determined statically.

- The programmer is required to state the scale (type) of the result.

```
G1 := G ( F1 * F2 );
```

Result of F1 * F2 (or 6.48 * F1, F1/8.6, F2/F1 etc) is <u>Universal Fixed</u>.

Arbitrarily high accuracy of Universal Fixed.

G reduces the accuracy to explicit type

Composite operation obeys the usual rules of the accuracy of real operations

What is the type of a numeric literal ?

Real literal - type Universal Real

Integer literal - type Universal Integer

Implicitly converted to type required by context

Literal expressions are expressions whose only operands are numeric literals, operations are the predefined ones.

Literal expressions are equivalent to writing the value instead.

```
I := 3_000_000 - 3_000_000;
     ------------------------
     a literal expression = 0
```

Hence CONSTRAINT_ERROR will not be raised

Operations on Universal Integer are just the same as all integer types.

Universal Real encompasses capability of both float and fixed point.

```
ADD  : constant := 1.0 + 3.0;
SUB  : constant := 6.0 - 8.0;
PLUS : constant := + 12.0;
NEG  : constant := - 8.4;
MULT1 : constant := 2 * 3.0;
MULT2 : constant := 4.0 * 10;
MULT3 : constant := 5.0 * 10.1;
DIV1 : constant := 10.0 / 2;
DIV2 : constant := 10.0 / 2.0;
EXP  : constant := 3.0 ** 2;
ABS1 : constant := ABS(6.0);
```

Relational operators as well

Careful with = and /=

Exercises

Are the following literal expressions?

(a) F'DIGITS + 10

(b) FX'DELTA / 10.0

(c) FX'LAST - 0.1

(d) 1 / FX'ACTUAL_DELTA

Which is more accurate (X: FX)

10.1 + 10.1 + X or 10.1 + X + 10.1

Use of Universal Real is both more accurate and more efficient (at run-time).

Blue's Algorithm for $\sqrt{\sum a_i^2}$

- Hard to avoid underflow/overflow
- Example of high quality algorithm
- Comparison with FORTRAN possible

Assume following context:

```
type REAL is digits D;
type VECTOR is array (INTEGER range <>) of REAL;
function SQRT (X: REAL) return REAL;
```

Specification of function:

```
function NORM (X: VECTOR) return REAL;
```

Article referenced should be studied for full details

Algorithm requires some values to be stored exactly.

This can be done by literal expressions in Ada

Examples

```
EB1 : constant := (REAL'EMAX + 1) / 2;
        -- uses EMAX even and type Universal Integer

B1 : constant REAL := 2.0 ** (-EB1);  -- model number

OVERFL: constant REAL := REAL'LARGE * S2M;
        -- can be calculated by compiler

RELERR : constant REAL := SQRT (REAL (REAL'EPSILON));
        -- must be calculated dynamically
        -- conversion necessary
```

The vector could be too big

```
N : constant INTEGER := X'LENGTH;

if N > 2 ** REAL'MANTISSA then
        raise CONSTRAINT_ERROR;
end if;
```

However, if INTEGER'LAST < 2 ** REAL'MANTISSA, the comparison will yield CONSTRAINT_ERROR directly (which it must not). However, in this case, the comparison is unnecessary since the condition is always FALSE.

Algorithm accumulates three separate sums of squares for the small, medium and large valued terms.

Afterwards, the sums are inspected to give the combined result

```
return ABIG * SQRT (1.0 + (ASML/ABIG) **2);
```

This calculates

```
SQRT ( ABIG **2 + ASML **2)
```

without unnecessary risk of overflow.

Instead of using REAL attributes, the algorithm could use the attributes of the hardware type IP.

REAL'LARGE becomes REAL'BASE'LARGE

Advantages

- Closer match to actual hardware capability
- Only one routine for each hardware type

Could also use machine-dependent attributes (outside model).

Fixed Point Approximation

Smooth functions can be approximated by power series

```
Y := A + B*X + C*X**2 + D*X**3;
```

This method of evaluation gives at least one extra rounding error on every additional term. Nested multiplication is better:

```
Y := ((D*X+C)*X+B)*X+A;
```

Algorithm is taken from Cody and Waite - see references.

Given FR : purely fractional fixed point type

```
G : FR ;  -- reduced argument for sine/cosine
Y : FLOAT;
```

Approximation:

```
Y := FLOAT(
        FR((
          FR((
            FR((
              FR(0.00066_60872 * G)
            - 0.01267_67480) * G)
          + 0.13332_84022) * G)
        - 0.66666_62624) * G)
        );
```

```
package SIN_COS is
   function SIN ( X: FLOAT) return FLOAT;
   function COS ( X: FLOAT) return FLOAT;
end SIN_COS;
```

Use of FLOAT just an illustration.

Algorithm notes sign, add $\pi/2$ for COS, then calls COMMON_PART.

COMMON_PART does

1) check argument not too large
2) performs range reduction
3) does fixed point approximation
4) converts back and changes sign if necessary.

Integer type used for argument reduction:

```
B: constant := FLOAT'MANTISSA;
type INT is range 0 .. 4 * 2 ** (B/2);
```

Examples of number declarations:

```
PI : constant := 3.14159_26535_89793_23846;
PI_DIV_2 : constant := PI/2;
ONE_DIV_PI : constant := 1.0/PI;
```

(use of number declarations preserves accuracy)

Argument reduction is difficult

To do

$$F := X - \text{Integer\_part\_of}(X/\pi) * \pi$$

without loosing unnecessary accuracy

This is done by

```
F := ((X1 - XN * C1) + X2) - XN * C2;
```

where $C1 + C2 = \pi$ to more than machine accuracy

```
C1 : constant := 8#3.1104#;
C2 : constant := -8.9089_10206_76153_73566_17E-6;
```

Conditional Compilation

Sizes of mantissa of FLOAT determines the number of terms needed in fixed point approximation. Rely upon compiler optimization to give required code. If FLOAT is too accurate, raise CONSTRAINT_ERROR.

```
if B <= 24 then
   --
elsif B <= 32 then
   --
elsif B <= 50 then
   --
elsif B <= 60 then
   --
else
   raise CONSTRAINT_ERROR;
end if;
```

(warning from compiler if CONSTRAINT_ERROR always raised)

Example of generics

- For any real type
- Exploit generality of text
- Appropriate for general purpose routines

```
generic
    type REAL is digits <>;
package COMPLEX_OPS is
    type COMPLEX is record
                        RE, IM : REAL;
                        end record;
    function "-" (X : COMPLEX) return COMPLEX;
    function ABS (X : COMPLEX) return REAL;
    function "+" (X, Y : COMPLEX) return COMPLEX;
    function "-" (X, Y : COMPLEX) return COMPLEX;
    function "*" (X, Y : COMPLEX) return COMPLEX;
    function "/" (X, Y : COMPLEX) return COMPLEX;
end COMPLEX_OPS;
```

User's Program

```
type R is digits 10;
package MY_C is new COMPLEX_OPS(R);
use MY_C;
U, V : COMPLEX := (1.0, 2.0);
W : COMPLEX := U * V; -- COMPLEX multiplication
```